# Verification by Reduction to Functional Programs

THÈSE N$^{O}$ 7636 (2017)

PRÉSENTÉE LE 25 AOÛT 2017

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ANALYSE ET DE RAISONNEMENT AUTOMATISÉS
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Régis William BLANC

acceptée sur proposition du jury:

Prof. P. Ienne, président du jury
Prof. V. Kuncak, directeur de thèse
Prof. Ph. Rümmer, rapporteur
Dr N. Bjørner, rapporteur
Prof. M. Odersky, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

"The best way to predict the future is to invent it."
— Alan Kay

To the memory of my father

# Acknowledgements

First, I would like to thank my advisor, Viktor Kuncak, for guiding me through my PhD studies. He trusted me when I had no clue about which research direction to take, and I am grateful he let me explore projects in this way. He always showed an enthusiasm for research, which was naturally motivating.

I also thank the members of my thesis jury, Nikolaj Bjørjner, Philipp Rümmer, Martin Odersky, and Paolo Ienne, for taking the time to review and judge my thesis. Their questions and constructive comments helped me improve the final version of my thesis.

Although it is true that only the PhD student types in the content of this manuscript, the results presented were made possible only through the help of several co-authors. I want to thank each of my co-authors throughout my short scientific career: Etienne Kneuss, Philippe Suter, Laura Kovács, Ashutosh Gupta, Bernhard Kragl, Thibaud Hottelier, and Thomas Henzinger. In particular, Laura Kovács was my first adviser on a research project, and her guidance has was valuable in convincing me to start my PhD research. She also hosted me at TU Wien for the Summer before beginning my PhD work, which led to productive research and ultimately a publication. Philippe Suter suggested the original idea for tackling verification of imperative programming, which is the seed that led to the work presented in my thesis. Finally, although not a co-author, David Novo gave me the possibility to work on a cool and interesting research project, and I thank him for that.

The LARA group has always been an enjoyable environment to work in, mostly thanks to the members' relaxed attitude that would transpire through our irregular group meetings. I thank all the present and past members of the lab, as they helped create this nice atmosphere: Ali, Andrej, Andrew, Etienne, Eva, Filip, Georg, Giulano, Hossein, Ivan, Jad, Manos, Marco, Mikaël, Nicolas, Pierre-Emmanuel, Philippe, Ravi, Romain, Ruzica, Sumith, Tihomir. I am especially grateful to Philippe and Etienne who built most of the original infrastructure on which my research is based, to Nicolas who created a better and faster infrastructure and ported my work to it, and to Marco who seriously experimented with the language's extensions I was building and who provided valuable feedback. I also want to thank an often forgotten member of the lab, Leon, for his hard and repetitive work, which has been essential in putting this thesis together. I also thank the members of the LAMP lab, for building this awesome language that is Scala and for being generally cool.

If I was able to focus on research work, it is also due to the help of Yvette Gallay and Sylvie Jankow who, both, protected me from the complex administrative layers of EPFL. I am very thankful to them. I also thank Fabien Salvi, who shared my *love* for Microsoft Windows and

was always able to fix unexpected issues. I thank Holly, who had the courage to read through my entire thesis and dig out each and every mistake related to my *usage* of English.
Because EPFL would be pretty boring without people to hang out with, I want to thank the friends I met there: Alevtina, for introducing me to *The Americans*. Alexandre, for playing my games and for helping me distill my ideas (and for being at EPFL for a longer time than me). Alina, for organizing nice hikes. Ana, for always caring about my side projects and for the many discussions. Gilles, for chatting about hockey and playing football. Hông-Ân, for passing on her apartment to me in Zurich. Laurent, for sharing startup tips. Lucas, for cool tech discussions over lunch and coffee. Manohar, for being a fan of Roger. Marina, for hiking with me and for supporting Stan. Miji, for being always happy to visit places. Sonia, for sharing classic Swiss food and for the many nice coffee breaks. Stefan, for sharing my passion for coffee. I also thank all my friends outside EPFL, they helped me disconnect from work. In particular, I thank Mani, my long-time and dearest friend.
Finalement, bien sûr, je remercie ma famille. Mon frère, qui ne m'a dérangé de mon travail que pour organiser des matchs de foot. Mon chat, qui m'a montré son affection par ses griffures et morsures dont le but était, j'en suis sûr, de me retenir vers lui. Mon père, qui m'a fait découvrir le plus beau sport du monde, sport qui m'a bien aidé pour décompresser du travail. Et surtout ma mère, qui a fait tellement de sacrifices pour me permettre d'arriver où je suis aujourd'hui. Merci.

*Lausanne, April 2017* R. B.

# Preface

> Thus, I continued development creating Pascal in 1970. From then on, my goal was to create a language that was scientifically clean, i.e. defined not in terms of a mechanism (or even a specific computer), but in terms of a mathematical structure of axioms and derivation rules. This turned out to be an elusive goal.
>
> Niklaus Wirth, 2014

The potential of software to transform our society is virtually limitless. But this *potential* comes at the cost of limitless *complexity* and the difficulty of building the software that is *right*. Given this complexity, the *potential* of software to transform our lives turns into a *threat* to transform our lives. We increasingly rely on software for life critical tasks to transport us to see our loved ones, to diagnose diseases and to predict critical events in the near future. Yet each of these software services might simply crash in a bizarre way and stop delivering a time-critical service, or cause us to make a wrong decision, with disastrous consequences.
Astonishingly, the predominant methodology for constructing software leaves many potential errors behind. There is belief that efficiency dictates the use of low-level languages, for which the application of formal methods is difficult. Ambitious researchers have set to demonstrate that it is possible to verify low-level C code as is written. This has resulted in impressive achievements, yet the effort that was needed provides evidence that existing languages are not the most efficient way to construct software with strong correctness guarantees. On the other side of the spectrum, first-order functional programming languages have been shown suitable for reasoning about programs but tempt the developers into writing complex higher-order code for which efficient execution and reasoning can be also difficult to achieve.
The thesis of Régis Blanc points in the direction of a more productive approach for constructing software with strong correctness guarantees. The approach includes a language, which is a fragment of Scala that includes both higher-order and imperative constructs, and is designed to avoid constructs that are difficult to verify. The language prevents many errors *by construction*: it is memory safe, and it does not automatically include null in the space of possible values of data structures. Crucially, the language also permits efficient translation into a functional

form, to which verification techniques can be applied. The translation is kept manageable thanks to the use of unique mutable fields for data structures. The translation has been used successfully to verify programs in this language. At the same time, the programs in this fragment execute using efficient in-place updates and can use the familiar syntax of mutable variables, assignments, and loops.

Verifying the resulting functional code is a non-trivial task to which the thesis also contributes. It sets a high standard for formal models of code by representing machine integers correctly using bitvectors and requiring the developers to use unbounded integers in source code to obtain properties of mathematical integers. An entire range of *run-time errors* and undesired behaviors can be checked using the tool, including overflows, array bounds, pattern matching errors. Most interestingly, the developers can specify and verify correctness properties using preconditions, postconditions, and invariants. A crucial component in verification is the constraint solving of certain formulas describing possible executions of programs. The work of Régis shows how to keep an architecture of such tool maintainable by building a layer that communicates with multiple SMT solvers. What is more, he shows that future SMT solvers could be written in high-level languages: he writes a full-fledged SAT solver in Scala, starting from a simple version and refining it to a solver incorporating most techniques used in production quality SAT solvers.

The case studies in the thesis show that the approach holds a promise for constructing verified applications, and illustrates the practical benefit of Scala that can deploy applications on a number of platforms, including JVM, JavaScript, and, most recently, native code.

The thesis thus brings us a step closer towards an elusive goal of building software that we can reason about and execute efficiently.

*Lausanne, April 2017*

Viktor Kunčak

# Abstract

In this thesis, we explore techniques for the development and verification of programs in a high-level, expressive, and safe programming language. Our programs can express problems over unbounded domains and over recursive and mutable data structures. We present an implementation language flexible enough to build interesting and useful systems. We mostly maintain a core shared language for the specifications and the implementation, with only a few extensions specific to expressing the specifications. Extensions of the core shared language include imperative features with state and side effects, which help when implementing efficient systems. Our language is a subset of the Scala programming language. Once verified, programs can be compiled and executed using the existing Scala tools.

We present algorithms for verifying programs written in this language. We take a layer-based approach, where we reduce, at each step, the program to an equivalent program in a simpler language. We first purify functions by transforming away mutations into explicit return types in the functions' signatures. This step rewrites all mutations of data structures into cloning operations. We then translate local state into a purely functional code, hence eliminating all traces of imperative programming. The final language is a functional subset of Scala, on which we apply verification.

We integrate our pipeline of translations into Leon, a verifier for Scala. We verify the core functional language by using an algorithm already developed inside Leon. The program is encoded into equivalent first-order logic formulas over a combination of theories and recursive functions. The formulas are eventually discharged to an external SMT solver. We extend this core language and the solving algorithm with support for both infinite-precision integers and bit-vectors. The algorithm takes into account the semantics gap between the two domains, and the programmer is ultimately responsible to use the proper type to represent the data. We build a reusable interface for SMT-LIB that enables us to swap solvers transparently in order to validate the formulas emitted by Leon. We experiment with writing solvers in Scala; they could offer both a better and safer integration with the rest of the system. We evaluate the cost of using a higher-order language to implement such solvers, traditionally written in C/C++.

Finally, we experiment with the system by building fully working and verified applications. We rely on the intersection of many features including higher-order functions, mutable data structures, recursive functions, and nondeterministic environment dependencies, to build concise and verified applications.

**Key words:** software verification, decision procedures, functional programming, imperative

programming, constraint solving.

# Résumé

Dans cette thèse, nous explorons des techniques pour le développement et la vérification de programmes informatique dans un langage de haut niveau et sûr. Nos programmes peuvent exprimer des problèmes sur des domaines infinis et sur des structures de données récursives et mutables. Nous présentons un langage d'implémentation suffisamment flexible pour créer des systèmes intéressants et utiles. Autant que possible, nous maintenons un langage partagé pour la description des spécifications et pour l'implémentation, avec seulement quelques extensions pour exprimer les spécifications. Ces extensions comprennent des fonctionnalités impératives avec des effets de bord, qui permettent l'implémentation de systèmes efficaces. Notre langage est principalement un sous-ensemble du langage Scala, et il suit minutieusement la sémantique du standard officiel. Les programmes peuvent donc être compilés et exécutés avec la distribution officielle de Scala.

Nous présentons des algorithmes pour vérifier des programmes écrits dans ce langage. Nous utilisons une approche en couche par couche, où, à chaque étape, un programme est réduit vers un programme plus simple. Pour commencer, nous purifions les fonctions en enlevant les effets de bord et en les remplaçant par des types de retours explicites. Durant cette étape, nous réécrivons toutes les opérations de mutations de structures de données en des opérations de clonages. Ensuite, nous traduisons les calculs basés sur un état local en des calculs purement fonctionnelles, et ainsi nous éliminons toutes traces de programmation impératives. Le langage terminal est un sous-ensemble purement fonctionnelle de Scala, sur lequel nous appliquons un algorithme de vérification.

Nous intégrons notre suite de transformations dans Leon, un système de vérification formel pour Scala. Nous vérifions le langage fonctionnelle en utilisant un algorithme déjà présent dans Leon. Le programme est encodé dans une formule de logique du premier ordre qui utiliseh une combinaison de théories et des fonctions récursives. Ces formules sont envoyées à une procédure de décision externe. Nous ajoutons à ce language à la fois les nombres entiers à précision infinie et les nombres entiers représenté sur 32 bits. L'algorithme prend en compte la différence de sémantique entre ces deux domaines, et le programmeur est seul responsable pour en faire un usage correct. Nous développons une interface réutilisable pour le standard SMT-LIB qui nous permet d'échanger des solveurs de manière transparente, de manière à valider les formules émises par Leon. Nous expérimentons avec l'implémentation de solveurs en Scala ; ils pourraient offrir une intégration meilleure et plus sûre avec le reste du système. Nous évaluons le coût d'utiliser un langage de haut niveau pour implémenter de tels solveurs, qui sont traditionnellement développés en C/C++.

Finalement, nous expérimentons avec le système en développant des applications vérifiées et entièrement fonctionnelles. Nous utilisons l'intersection de nombreuses fonctionnalités, en particulier des fonctions d'ordre supérieur, des structures de données mutables, des fonctions récursives, et de dépendances externes non déterministes, de manière à créer des applications concises et vérifiées.

**Mots-clefs :** vérification logicielle, procédure de décision, programmation fonctionnelle, programmation impérative, résolution de contraintes.

# Contents

# List of Figures

# Introduction

*Writing software is hard.* It was hard when writing arcane assembly code, and it is still hard today when juggling with class hierarchies, first-class functions, and advanced types. Programming languages went through a significant number of evolutions: from relatively low-level and verbose instructions, to a concise and expressive syntax that facilitates the definition and composition of abstractions.

Arguably, static and strong typing offers invaluable help in building correct programs. With type inference, the syntactic overhead of writing types can often be eliminated; and when it cannot, explicit typing still provides useful documentation. Despite the many safety guarantees that strong typing provides, some well-typed programs can still go wrong at runtime.

Formal verification goes beyond typing by checking additional properties statically. Ideally, a verified program would never fail at runtime and always do the correct thing. In practice, ensuring complete functional correctness can be very challenging and comes with some costs. From the programmer's side, the behavior — the specifications — of the program should be described, which can become very tedious as it tends towards completeness. On the other side, the verification system and its algorithms grow in complexity along with the precision of the properties to verify.

Many verification systems have been built over the years. They span from adding lightweight analysis on top of a production programming languages to designing an entirely new language from scratch. Tools that augment an existing infrastructure with static analysis sometimes offer the advantage of being "free" to use: we simply run it on the codebase, as a complement to the compiler, and it outputs a list of potential issues. A strong selling point for such tools is the ease of integration into an existing workflow, and the low barrier to entry. However, if the checks that are performed are limited to non-logical errors, such as memory violations or null dereferences, then the scope of errors that can be detected is rather limited. This approach is also more suited to lower-level languages (C, C++, Java) where the weak typing can lead to more potential errors.

A possible next step in the design space is to add a specification language and to let the programmer describe properties that should hold. It can be as simple as using runtime assertions, but checking them at compile time. Additionally, such tools can support annotations for

writing contracts and invariants. In the end, the implementation can still be executed by the underlying platform, and a layer can be added on top of the verification system to bolster static guarantees of the program.

At the other extreme, we can build a new programming environment from the ground up. This language can be designed to play well with static verification by reducing as many unsafe, or expensive to verify, features as possible. Besides the significant engineering cost to developing a new programming environment, this approach can also suffer from a difficult adoption due to the steep learning curve of a new, and likely complex, language.

In this thesis, we present our contributions to the development of a verification system that falls somewhere in the middle of this design space. Our system, Leon[1], has been under active development over the past six years, and is the product of contributions of many people [SDK10, SKK11, Sut12, BKKS13, KKKS13, KKK15, BK15, Kne16]. We build on top of the Scala programming language. Leon understands a subset of Scala and can prove the correctness of properties that are expressed in this subset. Scala, with its strong typing, already quite well fits the requirements of a friendly language for verification. We limit ourselves to a subset of the language, as some of the features of the language are quite advanced hence difficult to reason about. In order to ensure more efficient and feasible verification, we carefully select which features of the language to work with.

Leon brings strong guarantees of static types to the expressive power of tests and run-time checks. Having the same specification and implementation language takes advantage of the clear semantics of the underlying language and unifies two related concepts. For the programmer without a strong background in formal logic, being able to relate to familiar language constructs is very encouraging. Although not universally used, such approaches have been adopted in the past, most notably in the ACL2 system and its predecessors [KMM00b], which have been used to verify an impressive set of real-world systems [KMM00a].

We believe that our approach finds a sweet spot in the design space. Firstly, by using the existing Scala compiler and the Java Virtual Machine, programs verified by Leon can be executed, without any modification, on the underlying system. These programs can be compiled directly by the official Scala compiler, thus taking advantage of the many optimizations and tooling of the Scala platform; and specifications can be either checked dynamically or turned off entirely in order to reduce runtime overhead. Secondly, the limitation to a subset enables us to choose the combination of features that translates well to the verification algorithms. We are essentially designing a small verification-friendly core language, within a larger existing language.

By its focus on imperative programs, our thesis differs from previous work on Leon. A purely functional language presents several advantages: For one, it is more direct to translate into logical formulas for automated reasoning, and for two, the fact that every expression is pure

[1] https://github.com/epfl-lara/leon

make programs easier to manipulate. However, most real-world applications are likely to need stateful features, at the very least for handling input/output and system dependencies. The addition of state to the language enables us to develop more realistic applications, with user interactions that are properly modeled.

As the core infrastructure of Leon works with the assumption of a purely functional subset of Scala, we implement the imperative layer of the language by a reduction to the functional core language. The system performs several successive rewritings of the input program until all extensions, in particular imperative features, are eliminated and translated into an equivalent functional form.

## Contributions

In this thesis, we explore the design and implementation of a modern and expressive language for building verified software. To be more amenable to verification, the language is designed to be largely compatible with Scala, but is slightly restrictive. It also extends Scala with additional notations for describing specifications.

- We describe the Leon input language by providing a comprehensive grammar and an expansive discussion of its semantics and restrictions. The grammar describes a proper subset of Scala programs, and the accepted programs follow the typing derivations from Scala. The language restricts the sharing of pointers to mutable objects, in a sense that we specify in this thesis. We complement the formal presentation of the language with an extended tutorial that illustrates, in a more interactive way, how Leon can be used to develop safe software.

- We discuss how to integrate objects with state into the Leon framework. We present an algorithm that rewrites functions with effects on such objects into functions with a pure signature. To simplify the rewriting, the translation generates local assignments hence does not eliminate local state. The algorithm makes a global program transformation to modify the signatures of all functions to a pure signature. We also justify the introduction of restrictions on aliasing.

- We show how to eliminate local state by reducing it into purely functional expressions. We take advantage of statically having access to the entire scope hence of being able to perform more advanced transformations. In particular, local functions can access and update local state, as long as the function does not escape the scope of its parent definition. We explain why these functions, with implicit effects, can only be supported in a closed environment.

- We encode fundamental Leon types in a sound way. In particular, we introduce a distinction between bit-vectors and mathematical integers, and to give access to both in the language, we expose an API. We show how to represent Scala arrays and how to

ensure that their implicit invariants are respected. We automatically generate a number of verification conditions that checks the validity of important safety properties when using these data types, such as detecting division by zero and correctly indexing in the proper bounds of arrays. We also, optionally, generate verification conditions that detect potential overflows when performing arithmetic on bit-vectors.

- The aforementioned individual techniques and algorithms are implemented and integrated into the Leon verification system. The implementation is documented and was extensively tested; and the whole system is beginning to be used outside of our research group. The features we contribute in this thesis help reduce the gap with production Scala code. It does so by bringing familiar and important idioms to the language.
- At the core of the Leon infrastructure, SMT solvers do the heavy lifting of automatically proving logical formulas. An efficient SMT solver is mandatory for the proper functioning of Leon. State-of-the-art solvers are stand-alone packages, usually written in a non-managed language such as C, and they must thus be interfaced with Leon, implemented in Scala. To facilitate the communication with solvers, we develop a Scala library to wrap the SMT-LIB standard format. Using this library, we are able to make Leon solver-independent. We also experiment with our own solver infrastructure that is directly implemented in Scala. We develop a modern SAT solver and we compare its performance with native solvers.
- Finally, to demonstrate the potential and expressive power of Leon, we look at the development of larger applications. These applications rely on system dependencies and user interactions, which make them more challenging to model properly. We can prove the validity of a significant number of properties on these benchmarks, and we can compile them with the standard Scala compiler and run them on the Java Virtual Machine, with the guarantee that the verified properties will hold.

## Outline

The rest of this thesis is organized as follows:

**Chapter 1** introduces the Leon language with an extended tutorial. The tutorial demonstrates how to develop correct and safe software with Leon. We then document the whole input language, with a focus on the specific features introduced in this thesis. We provide a formal grammar for the input language. We discuss the design decisions made while deriving the exact features and trade-offs of the language. We conclude this chapter with an overview of the architecture of the Leon system.

**Chapter 2** presents the transformation rules for reducing local imperative code into purely functional expressions. We use the fact that the entire scope is known ahead of time to perform a complete and sound transformation. The output of this phase is a purely functional program.

**Chapter 3** presents the reduction of functions with effects on mutable types to pure functions; these pure functions return the new values instead of mutating its parameter. The transformation phase also eliminates all mutation operations on objects, and replaces them by local assignments of explicit copies.

**Chapter 4** covers the solving infrastructure of Leon. We give some background on solving formulas with recursive functions, which is at the core of Leon [SDK10, SKK11]. Then, we explain how we encode some of the primitive data types in Leon, including numbers and arrays. We present a library that we developed to abstract away SMT solvers. Finally, we discuss a SAT solver implementation that is in pure Scala, and how it compares with a reference implementation in Java.

**Chapter 5** experiments with the Leon system and its new capabilities. We develop some more significant benchmarks, involving user interactions and system dependencies. We report on the experience of using Leon to develop and verify such programs.

**Chapter 6** discusses related work and how it compares to the techniques developed in this thesis.

# 1 An Overview of Leon

In this chapter, we present the programming language currently supported by Leon. We introduce the language in the form of a tutorial, showing how to implement and verify basic programs with Leon. We then define the precise syntax of the language, as well as the typing restrictions. We discuss the design decisions and trade-off that led to the current state of the language. We conclude this chapter with an overview of the Leon architecture, setting the stage for the following chapters. The content of this chapter is intentionally presented as a high-level tour of Leon and can be used as a user's manual. Our scientific contributions are detailed in Chapter 2, Chapter 3, and Chapter 4.

Leon's language is a modern and expressive language that mixes imperative programming with higher-order functions and recursion. It relies on a statically global assumption for aliasing, which enables programs to be amenable efficiently to verification.

The Leon language is essentially a subset of Scala, with a few extensions. To maintain compatibility with the official Scala compiler, the extensions are implemented as a library. Some of the extensions, notably a limited number of the specification facilities, provide only a placeholder implementation in the library, thus are parsable by the standard Scala compiler but not executable with the exact same semantics.

The language mostly maintains a shared specification and implementation language, but some features are restricted in how they can be used. Side-effects, for example, cannot be used when writing specifications. A small number of features are used exclusively for writing specifications. We formalize the language of specifications later in this chapter.

Leon has been under development for more than five years. The system has an official documentation[1] that covers, among other things, most of the features presented in this thesis. We have also contributed additional documentation in the form of Scaladoc, available with the Leon source code[2]. Leon is extensively tested, with an integration server that automatically

[1] http://leon.epfl.ch/doc/
[2] https://github.com/epfl-lara/leon

runs a complete test suite on each build. We have contributed more than 200 individual test programs, as well as more than 2000 lines of unit tests to the current test suite.

## 1.1 Tutorial: Developing Correct Software with Leon

We introduce the flavor of verification and error finding in Leon through data structure and algorithm examples. The online interface at http://leon.epfl.ch/ provides us the opportunity to test the system and its responsiveness. The complete sources for the examples we develop in this section are available in Appendix A.1.

**Algebraic Data Types** In Leon (and Scala), algebraic data types are defined with the **case class** construct. Let us start with the implementation of a fundamental functional data structure:

```
abstract class List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
case class Nil[A]() extends List[A]
```

We define the List data type as being either a Cons of an element and another List, or the empty list Nil. Our definition is generic in the type A, so we will be able to define generic functions and apply them to any concrete type. People familiar with Scala might notice a difference with the idiomatic definition of Nil, which should be

```
case object Nil extends List[Nothing]
```

but Leon does not have a notion of general subtyping.

Along with the list structure, we should also provide basic functionalities so that clients can work productively with List. One very common operation is the size function:

```
def size[A](l: List[A]): Int = l match {
  case Nil() => 0
  case Cons(_, t) => 1 + size(t)
}
```

The implementation uses the convenient technique of pattern matching to manipulate algebraic data types. The implementation is then a simple recursion on the list. Leon can already start to verify this minimal program. If we were to run it, we would see that Leon verified the exhaustiveness of pattern matching. Out of the box, Leon will check several correctness properties of the program without the need of any particular annotations. We will come back in Chapter 4 to what properties are automatically ensured in Leon programs, but as a start let us add a simple postcondition to the function size. In Leon, we support the **ensuring** keyword, which takes a predicate as an argument and checks it against the receiving expression. It works

as follows:

```
def size[A](l: List[A]): Int = (l match {
  case Nil() => 0
  case Cons(_, t) => 1 + size(t)
}) ensuring(res => res >= 0)
```

The **ensuring** is applied on the body of the function and asserts that the result is always positive. Note that existence of **ensuring** is not Leon-specific and works with the standard Scala compiler. The **ensuring** construct is defined in the Scala standard library and is implemented to check the predicate at runtime. However, Leon will extract it and attempt to prove the property *statically*.

Regarding the size contract, it seems trivially correct, hence we expect Leon to prove its correctness. But, the situation is not that simple, and Leon does not prove the validity of the postcondition which suggests that something is actually wrong with the code. And something is indeed wrong, as size does not take into account possible overflow of the int type. A list of size longer than Int.MaxInt could indeed lead to a negative result.[3]

How do we fix this issue? Well, there is no easy fix to avoid the overflow with int, as the size of the list is unbounded. The mistake is in the signature of size, that should not be returning an int, but should be returning a BigInt — an infinite precision integer.

```
def size[A](l: List[A]): BigInt = (l match {
  case Nil() => BigInt(0)
  case Cons(_, t) => 1 + size(t)
}) ensuring(res => res >= 0)
```

With this new implementation, Leon is able to prove the correctness of the postcondition. In this small example, we see that Leon is very precise with the semantics of primitive types and operations, and it always maintains soundness. This is an important feature of Leon and our modeling of Scala, which we will detail in Chapter 4; we will examine additional manifestations of our decisions later. In particular, the Int primitive type is treated as a true bit-vector of size 32, and we use the BigInt type from the standard Scala library to model mathematical integers.

The size implementation above is not tail recursive, which could lead to performance and memory usage less optimal than a tail recursive one. When offering a library, we generally need to provide an optimized implementation. A tail-recursive implementation, coupled with the proper optimizations in the compiler would serve this purpose, but alternatively we could simply write an imperative version without using recursion.

[3] Another potential issue with the above is linked to memory, in particular the stack's size might not be able to support a recursion on such a long list. In this work, we will put aside the question of how to check out-of-memory errors, and we will assume memory is unlimited.

```
def size[A](l: List[A]): BigInt = {
  var res: BigInt = 0
  var lst: List[A] = l
  while(!isEmpty(lst)) {
    lst = tail(lst)
    res += 1
  }
  res
}
```

Leon supports the fundamental building blocks of imperative programming, assignments, sequence of instructions and looping. Here, we decided to use two new functions, isEmpty and tail, to iterate through the list. We could have used pattern matching, but combining pattern matching and conditions in loops tend to be a bit awkward to write, and using isEmpty and tail is a common pattern when manipulating lists. However, we need to define these functions, and tail, in particular, should be defined only on Cons instances.

```
def isEmpty[A](l: List[A]): Boolean = l match {
  case Nil() => true
  case _ => false
}
def tail[A](l: List[A]): List[A] = l match {
  case Cons(_, t) => t
}
```

This code will compile but will result in a match-exhaustiveness counterexample reported by Leon. Indeed, tail is intended as a global function as part of the API of List, and for this reason even though it looks like size makes a safe usage, it is not safe to assume the argument could not be an empty list in general. Note that declaring the parameter of tail as a Cons (instead of a List) does not work, as the type of lst is List. The solution here is to introduce the notion of preconditions.

In Leon, to define a precondition to a function, we use the keyword **require**. A precondition is a Boolean expression that involves the parameters of the function, and that the caller must ensure when invoking the function. **require** is part of the standard Scala distribution and works similarly to an assertion, with a runtime crash if the expression evaluates to false. Applying this to tail, we obtain the following:

```
def tail[A](l: List[A]): List[A] = {
  require(!isEmpty(l))
  l match {
    case Cons(_, t) => t
  }
}
```

Notice a few interesting things happening now. First, we still have the missing case in the pattern-matching expression, however Leon no longer indicates a match-exhaustiveness error, as it is now able to use the precondition to determine that the program will never take the path through the missing pattern. In contrast, the standard Scala compiler will still issue a warning for the missing case. Second, we are able to make a call to another user-defined function in the precondition (isEmpty); this demonstrates that the language of specifications is essentially as expressive as the language of implementation. There are some limitations on which features can be used for specifications, we will formalize these later, but most of the limitations involve the usage of state. Essentially, any purely functional piece of code can be used as a specification.

Leon will use a precondition as an assumption to the execution of a function, hence any correctness property of a function, such as match exhaustiveness or postcondition is valid assuming the precondition is respected at each call site. In order to ensure this, Leon will additionally checks the precondition at every call site, using assumptions at the caller scope. In particular, the size implementation properly uses tail, which Leon can prove due to the loop condition guarding the call.

With the combination of preconditions and postconditions, Leon supports a programming style known as design by contracts [Mey86, Ode10]. The developer organizes most of the code as functions, each with their own contract that other functions must respect. Leon then verifies the contract of each function, that is to say for any inputs that respect the precondition, then the postcondition must hold. Finally, Leon checks that at all call sites, preconditions are properly respected. The contracts act as formal documentation of functions, stating what the caller must respect, and what it can expect from the result.

Currently, Leon checks that size respects the contract of tail, but we should also add a proper contract to the size function itself. We could check the same postcondition as with the functional implementation, but it would be better to check equivalence with the functional implementation. It can be argued that the functional implementation is simple and close enough to the definition of the operation, that it could serve as a specification for the imperative implementation that is more focused on how to do the operation. Therefore we can rename the original implementation as sizeSpec and use it in the contract of size:

```
def sizeSpec[A](l: List[A]): BigInt = l match {
  case Nil() => BigInt(0)
  case Cons(_, t) => 1 + size(t)
}
```

```
def size[A](l: List[A]): BigInt = {
  var res: BigInt = 0
  var lst: List[A] = l
  while(!isEmpty(lst)) {
    lst = tail(lst)
    res += 1
  }
  res
} ensuring(res => res == sizeSpec(l))
```

The above contract states essentially that both implementations are equivalent, and if we consider the functional implementation as the full specification of the size operation, then we are essentially trying to check functional correctness of size, as opposed to just checking some properties, such as being positive.

However, Leon is not able to prove the above without additional help. There is no bug, but the presence of a loop without an invariant complicates the problem and Leon will time out trying to unroll the loop. To make the proof go through, the programmer needs to provide a loop invariant strong enough to prove the postcondition. The following is sufficient:

```
import leon.lang._

def size[A](l: List[A]): BigInt = {
  var res: BigInt = 0
  var lst: List[A] = l
  (while(!isEmpty(lst)) {
    lst = tail(lst)
    res += 1
  }) invariant(res + sizeSpec(lst) == sizeSpec(l))
  res
} ensuring(res => res == sizeSpec(l))
```

The syntax for loop invariant is the **invariant** method that can be appended to a **while** loop. The parentheses are needed, in order to obtain a syntax compatible with Scala, we take advantage of some implicit conversion definitions. In particular, we define **invariant** as a library, hence the **import** in the first line. The invariant is not checked at runtime by our implementation, but is always checked statically by Leon. Leon will prove independently that the invariant holds at initialization (when first entering the loop), and that it holds at iteration (the inductive case). On loop exit, to derive the postcondition, Leon will use the invariant in conjunction with the negation of the loop condition.

**Mutable Objects** Up until now, our use of side-effects was kept local, hence the functions still appear pure to the outside world. Leon also supports functions with side-effects, as long as the modified objects are passed explicitly to the functions and not referred from the global

scope. Let us consider a simple representation of a bank account:

```
case class BankAccount(var checking: BigInt, var savings: BigInt)
```

A bank account has an intrinsic identity, and modeling it as a mutable object is a reasonable choice. In Leon, we introduce mutable objects by marking some fields as **var**, just as in Scala. Hopefully, bank accounts never go in the red, and the particular bank we are modeling does not permit negative balance, so we can use the class invariants to express that constraint:

```
case class BankAccount(var checking: BigInt, var savings: BigInt) {
  require(checking >= 0 && savings >= 0)
}
```

We use **require** for class invariants as well, but note that the semantics is stronger than in Scala: it is checked whenever an object is mutated, and not only at initialization. Finally, we can provide a balance function that sums up the information of the account:

```
case class BankAccount(var checking: BigInt, var savings: BigInt) {
  require(checking >= 0 && savings >= 0)
  def balance: BigInt = {
    checking + savings
  } ensuring(_ >= 0)
}
```

It is defined as a method of the class, and we can already verify that it is always positive. We can start defining operations on this bank account, still as methods:

```
  def save(amount: BigInt): Unit = {
    require(amount >= 0 && amount <= checking)
    checking = checking - amount
    savings = savings + amount
  } ensuring(_ => balance == old(this).balance)
```

The save operation transfers some amount of money from the checking to the savings entry. It modifies the bank account in-place, hence why it returns Unit and updates the fields directly. The precondition is essential, as omitting it would fail several invariants. First, subtracting from checking could lead to a negative value, breaking the class invariant. Next, adding a negative amount to savings could also lead to a negative value. And finally, saving a negative value is not really saving anyway. The postcondition states that the account is only moving money internally; and as Leon verifies it, both the bank and the customers should be able to sleep peacefully.

Notice our use of BigInt for all quantities in order to avoid overflow. If we were to use Int, Leon would detect several violations, reminding us that finite precision integers are not a good choice for banking operations. The reader would certainly not want to save some money and

end up with a negative account at the end of the day.

This example introduces a new notation for the postcondition, the old construct. It is a special specification-only notation, that refers to an identifier in the state it was when entering the function. In our example, we refer to the value of **this** account, before performing the operation, and comparing the balance at that point to the new balance in the end. Leon does not actually provide a runtime implementation for old, as it would need to perform a deep copy of the object when entering the function. The intended use is that a verified postcondition would be removed statically, and the program would be able to execute. The Leon library also provides new definitions for **require** and **ensuring**, in the package leon.lang.StaticChecks; these contracts are not checked at runtime, but will still be verified statically.

As an example of a more complicated operation, the following implements a transfer between the checking entries of two accounts:

```
def transfer(from: BankAccount, to: BankAccount, amount: BigInt): Unit = {
  require(amount >= 0 && from.checking >= amount)
  from.checking -= amount
  to.checking += amount
} ensuring(_ => from.balance + to.balance == old(from).balance + old(to).balance &&
                from.checking == old(from).checking - amount &&
                to.checking == old(to).checking + amount)
```

The precondition ensures that class invariants are not violated; the postcondition checks that no money has been magically lost, and that the accounts have the proper checking value after performing the operation. One global property enforced by Leon is that, on entering the scope of a function, no two pointers refer to the same mutable object. In this particular example, Leon is able to assume that from and to are two distinct accounts, which is a necessary condition to prove the postcondition.

Here is a good time to discuss the notion of equality. In Scala, equality can either refer to reference equality or structural equality. Reference equality checks that the objects are actually the same, in memory. Structural equality, for case classes, checks equality of each field. Primitive types are stored by value, so equality is solely value equality. For case classes, the default == operator performs structural equality on the fields declared in the main constructor of the case class. Interestingly, in Scala, the following would not raise an assertion violation:

```
case class A(a: Int) {
  var b: Int = 0
}
val a1 = A(0)
val a2 = A(0)
assert(a1 == a2)
a2.b = 42
assert(a1 == a2)
```

This occurs because the default implementation of equality on case classes compares only the parameters list of the constructor. In Leon, we do not permit the definition of **var** fields outside of the parameter list, which avoids that specific issue altogether. Scala has a reference equality operator eq. Leon only supports == for structural equality on case classes, and value equality on primitive types. Leon does not implement an eq operator to check reference equality between two objects.

**Higher-Order Functions** We can also combine state and side-effects with higher-order functions. For example, let us represent the notion of a transaction that stores a delayed, primitive bank operation, such as debiting or crediting an account:

```
case class Transaction(
  operation: (BankAccount, BigInt) => Boolean,
  cancel: (BankAccount, BigInt) => Unit,
  account: BankAccount, amount: BigInt, var executed: Boolean
) {
  def execute(): Boolean = {
    require(!executed)
    executed = operation(account, amount)
    executed
  }
  def rollback(): Unit = {
    require(executed)
    cancel(account, amount)
    executed = false
  }
}
```

We model an operation as a function that takes a bank account and an amount of money and that returns the status of the operation (whether it was successfully executed or not). We expect that the operation will have a side-effect on the account when invoked, but only if successful. As we would like to be able to rollback a transaction, we must also provide a way to cancel the operation.

Our implementation keeps an internal state about whether it has been executed or not, and uses preconditions to its methods to ensure proper usage. We assume that operations can fail, for example if the account does not have enough money left, or in case of network failure. With these primitives, we can implement a generic way to handle any transaction, such as retrying until it succeeds:

```
def retry(transaction: Transaction, times: Int): Boolean = {
  require(times > 0 && !transaction.executed)
  var i = 0
  var success = false
  (while(i < times && !success) {
    success = transaction.execute()
    i += 1
  }) invariant(i >= 0 && i <= times && success == transaction.executed)
  success
} ensuring(status => status == transaction.executed)
```

We can also implement the atomic execution of two transactions:

```
def execute(transaction1: Transaction, transaction2: Transaction): Boolean = {
  require(!transaction1.executed && !transaction2.executed)
  if(transaction1.execute()) {
    if(transaction2.execute()) true else {
      transaction1.rollback()
      false
    }
  } else false
} ensuring(executed => (
  (executed ==> (transaction1.executed && transaction2.executed)) &&
  (!executed ==> (!transaction1.executed && !transaction2.executed))
))
```

The postcondition ensures that either both, or neither, transactions are executed. The precondition checks that execute are called only with transactions that are still waiting for execution.

We define two checked operations of credit and debit; they fail without changing the account under some conditions:

```
def addOp(acc: BankAccount, amount: BigInt): Boolean = {
  if(amount < 0) false else {
    acc.checking += amount
    true
  }
}
def subOp(acc: BankAccount, amount: BigInt): Boolean = {
  if(amount < 0 || amount > acc.checking) false else {
    acc.checking -= amount
    true
  }
}
```

Using these, we can re-implement the transfer operation as a combination of execute and the

primitive operations:

```
def transfer2(from: BankAccount, to: BankAccount, amount: BigInt): Unit = {
  require(amount >= 0 && from.checking >= amount)

  val status = execute(
    Transaction((acc, amount) => subOp(acc, amount),
                (acc, amount) => addOp(acc, amount),
                from, amount, false),
    Transaction((acc, amount) => addOp(acc, amount),
                (acc, amount) => subOp(acc, amount),
                to, amount, false)
  )
  assert(status)
} ensuring(_ =>
    from.balance + to.balance == old(from).balance + old(to).balance &&
    from.checking == old(from).checking - amount &&
    to.checking == old(to).checking + amount
  )
```

Notice that Leon is able to prove that the status is always successful, due to the precondition and the specific instantiations of transactions and operations.

When combining state and higher-order function, one common idiom is to create closures that capture local state. A typical example is the concise implementation of a counter in Scala:

```
def makeCounter: (Unit) => Int = {
  var c = 0
  () => {
    c += 1
    c
  }
}
val counter = makeCounter
counter() //returns 1
counter() //returns 2
```

In Leon, functions do not carry an environment along with them. A function can only modify a value that it receives explicitly as an argument. A counter returned by makeCounter has type (Unit)=> Int, hence is not able to have side-effects in our system. The above example is rejected by Leon because the lambda function captures a variable.

Although this seems limiting, there is way to encode such closures in a concise way, which still enables implementing generic combinators with higher-order functions. We first define the notion of a stateful function:

```
case class State[A](var value: A)
case class SFun[A, S, B](state: State[S], f: (A, State[S]) ⇒ B) {
  def apply(a: A): B = f(a, state)
}
```

The SFun class is essentially a function from A to B, with an additional state parameter S. The state can be any type, including tuples and other complex case classes. Once created, using an instance of SFun is no different than using a closure such as the counter above, due to the apply method that handles the internal state.

When creating a stateful function, the difference with regular Scala closures is that there is no capture of local variables; rather the explicit initial state is passed and received at each invocation of the closure. The makeCounter example becomes

```
def makeCounter: SFun[Unit, BigInt, BigInt] = {
  SFun(State(0),
       (_: Unit, c: State[BigInt]) ⇒ {
         c.value += 1
         c.value
       })
}
val counter = makeCounter
counter() //returns 1
counter() //returns 2
```

Note that only the creation of closure inside makeCounter has changed slightly. To illustrate this, an SFun can essentially be used just as a regular closure. We finally present the implementation of a generic foreach function over the List type we developed:

```
def foreach[A, S](l: List[A], sf: SFun[A, S, Unit]): Unit = l match {
  case x::xs ⇒
    sf(x)
    foreach(xs, sf)
  case Nil() ⇒
    ()
}
```

In order to test it, we add, for convenience, the definition of the :: operator in the List class, and then we can use foreach as follows:

```
def testForeach(): Unit = {
  val l: List[Int] = 1::2::3::4::5::6::7::8::9::10::Nil[Int]()
  val sum = State(0)
  foreach(l, SFun[Int, Int, Unit](
                sum, (el: Int, s: State[Int]) => s.value += el))
  assert(sum.value == 55)
}
```

## 1.2 Input Language

In this section, we present a comprehensive grammar for the syntax of the Leon input language, which acts as a formalization of what subset of Scala is supported by Leon. The keywords defined as built-in in the grammar but not part of the Scala language, are emulated in Scala by library definitions.

This technique — emulating a natural syntax via a library — is a fairly common approach in Scala, which is made possible by the powerful and flexible syntax of the language. To introduce all the specific extensions needed for verification, we use a combination of implicit conversions and standard library definitions. Leon then ignores the implementation and handles them specifically to check contracts and specifications. The advantages of maintaining the compatibility with Scala is that we get to rely on several phases from the Scala compiler, including type checking and implicits resolution, and we are able to compile and execute a Leon program with the standard Scala compiler.

### 1.2.1 Syntax

As in Scala, the number of white spaces or carriage returns is not important, as long as the lexer is able to differentiate two tokens from each other. The one exception to this rule is when using new lines to infer semicolons. In the grammar, we use a semicolon every time a semicolon is technically needed. But in practice, most of them will be inferred by the Scala front-end and are thus optional. Additionally, the grammar does not describe all the syntactic sugar expanded by Scala, although it will be compatible with our system as it shares part of the same front-end. For example, notation such as `x += 1` are extracted in Leon as `x = x.+(1)`, the latter being a valid expression from the grammar. In a similar fashion, infix notation is handled by the syntactic sugar transformation into a method invocation.

**Definitions** Figure 1.1 describes the top level structure of a Leon program, which is essentially a sequence of modules — Scala **object**s — each containing a combination of data structures and function definitions. This structure works well in a design by contract philosophy, where each function exposes a contract and is available from the top-level scope.

⟨ *program* ⟩ ::= ⟨ *unit* ⟩$^+$
⟨ *unit* ⟩ ::= `object` ⟨ *identifier* ⟩ `{` ⟨ *definition* ⟩$^*$ `}`
| ⟨ *class_def* ⟩
⟨ *class_def* ⟩ ::= `abstract class` ⟨ *identifier* ⟩ ⟨ *type_parameters* ⟩? ⟨ *class_body* ⟩?
| `case class` ⟨ *identifier* ⟩ ⟨ *type_parameters* ⟩? `(` ⟨ *field_decls* ⟩? `)`
⟨ *class_extend* ⟩? ⟨ *class_body* ⟩?
| `case object` ⟨ *identifier* ⟩ ⟨ *class_extend* ⟩? ⟨ *class_body* ⟩?
⟨ *definition* ⟩ ::= ⟨ *class_def* ⟩
| ⟨ *fun_def* ⟩
⟨ *class_extend* ⟩ ::= `extends` ⟨ *identifier* ⟩
⟨ *field_decls* ⟩ ::= ⟨ *field_modifier* ⟩? ⟨ *identifier* ⟩ `:` ⟨ *type* ⟩
| ⟨ *field_decls* ⟩ `,` ⟨ *field_modifier* ⟩? ⟨ *identifier* ⟩ `:` ⟨ *type* ⟩
⟨ *field_modifier* ⟩ ::= `val` | `var`
⟨ *type_parameters* ⟩ ::= `[` ⟨ *type_param_list* ⟩ `]`
⟨ *type_param_list* ⟩ ::= ⟨ *identifier* ⟩ ⟨ *mutable_type* ⟩?
| ⟨ *identifier* ⟩ ⟨ *mutable_type* ⟩? `,` ⟨ *type_param_list* ⟩
⟨ *mutable_type* ⟩ ::= `:Mutable`
⟨ *param_decls* ⟩ ::= ⟨ *identifier* ⟩ `:` ⟨ *type* ⟩
| ⟨ *param_decls* ⟩ `,` ⟨ *identifier* ⟩ `:` ⟨ *type* ⟩
⟨ *implicit_decls* ⟩ ::= `(implicit` ⟨ *param_decls* ⟩ `)`
⟨ *class_body* ⟩ ::= `{` ⟨ *precondition* ⟩? ⟨ *fun_def* ⟩$^*$ `}`
⟨ *fun_def* ⟩ ::= `def` ⟨ *identifier* ⟩ ⟨ *type_parameters* ⟩? `(` ⟨ *decls* ⟩ `)` ⟨ *implict_decls* ⟩?
`:` ⟨ *type* ⟩ `=` ⟨ *fun_body* ⟩
| `val` ⟨ *identifier* ⟩ `=` ⟨ *expr* ⟩
⟨ *fun_body* ⟩ ::= ⟨ *expr* ⟩ | `{` ⟨ *precondition* ⟩? ⟨ *expr_seq* ⟩ `}` ⟨ *postcondition* ⟩?
⟨ *precondition* ⟩ ::= `require (` ⟨ *expr* ⟩ `) ;`
⟨ *postcondition* ⟩ ::= `ensuring (` ⟨ *lambda* ⟩ `)`

Figure 1.1 – Grammar for the definitions of the Leon input language.

Class can have definitions, but their main use is to represent algebraic data types, with the **case class** pattern. For convenience, it is possible to add definitions in the body of a class, however Leon is not equipped with full object-oriented reasoning, and methods are essentially lifted outside as functions taking the receiver object as a parameter. In particular, inheritance is limited as abstract classes do not have constructor parameters, hence children classes cannot pass their parameters to the parent. Besides, there is no **override** keyword to override method definitions.

It is important to understand that state can be introduced only in a very controlled way, that is, by prepending a class parameter with a **var** modifier. This turns the class definition into a mutable class that can then be mutated in place, when received as an argument in functions. There are no global variable definitions; only **val** and **def** can be defined at a module level. It is also not possible to define a global **val** containing an object of mutable type. This means that Leon does not have global state, all state has to be explicitly passed by reference from function to function. A common pattern to do so is to use implicit parameters, we discuss this technique further in Chapter 3. Even though introducing global state would be conceptually easy, as described in Section 3.4.1, we believe this feature to be harmful for both maintenance and reasoning about programs, while not being essential.

Although Leon supports **val** definitions at both the module and the class level, we represent **val**s as function definitions with zero parameters. We can afford such a representation because we prevent global **val**s from pointing to mutable objects. We do so only for module-level and class-level definitions; local **val**s are preserved.

Leon supports natively Set and Map, very common data structures that are useful both for implementation and specifications. Even though they are defined as a library in Scala, they are a native data structure in Leon and are handled as first-class citizen in the solver. Leon also supports Array natively. Arrays are the primary data structure for writing imperative programs, and we support all the basic operations: initializing a fixed size array with default elements, in-place update, and random access.

Array is the only primitive type in Leon that is mutable. All other mutable types come from class definitions with variable fields. This also means that, along with the field assignment operator, the array update operation is the only primitive operation that can modify the value of an object. We make a distinction here with local **var**s that are also supported; but they are only a local state in the current scope and not properties of objects.

**Expressions** Figure 1.2 formalizes the valid syntax for building expressions in Leon. As with definitions, we restrict Scala to a subset that we are able to manage properly all the way through verification.

We see the second way that state can be introduced, which is locally to an expression — a code block in most practical cases. Thus, function can make use of variables locally and can

⟨ *expr_seq* ⟩ ::= ⟨ *expr* ⟩ | ⟨ *expr* ⟩ `;` ⟨ *expr_seq* ⟩
⟨ *expr_list* ⟩ ::= ⟨ *expr* ⟩ | ⟨ *expr* ⟩ `,` ⟨ *expr_list* ⟩
⟨ *lambda* ⟩ ::= ⟨ *param_decls* ⟩ `=>` ⟨ *expr* ⟩
⟨ *expr* ⟩ ::= ⟨ *numeral* ⟩ | `true` | `false` | `()` | ⟨ *identifier* ⟩
| `(` ⟨ *expr* ⟩ `,` ⟨ *expr_list* ⟩ `)`
| `if` `(` ⟨ *expr* ⟩ `)` ⟨ *expr* ⟩ `else` ⟨ *expr* ⟩ | `if` `(` ⟨ *expr* ⟩ `)` ⟨ *expr* ⟩
| `{` ⟨ *expr_seq* ⟩ `}`
| ⟨ *fun_def* ⟩ `;` ⟨ *expr* ⟩
| `var` ⟨ *identifier* ⟩ ⟨ *type_ascript* ⟩? `=` ⟨ *expr* ⟩ `;` ⟨ *expr* ⟩
| `val` ⟨ *identifier* ⟩ ⟨ *type_ascript* ⟩? `=` ⟨ *expr* ⟩ `;` ⟨ *expr* ⟩
| ⟨ *identifier* ⟩ `=` ⟨ *expr* ⟩
| ⟨ *identifier* ⟩ `(` ⟨ *expr* ⟩ `)` `=` ⟨ *expr* ⟩
| ⟨ *identifier* ⟩ ⟨ *instantiation* ⟩? `(` ⟨ *expr_list* ⟩? `)`
| `assert` `(` ⟨ *expr* ⟩ `)`
| ⟨ *expr* ⟩ `match` `{` ⟨ *case* ⟩$^+$ `}`
| ⟨ *expr* ⟩ `.` ⟨ *identifier* ⟩
| ⟨ *expr* ⟩ `.` ⟨ *identifier* ⟩ ⟨ *instantiation* ⟩? `(` ⟨ *expr_list* ⟩? `)`
| ⟨ *expr* ⟩ `.` ⟨ *identifier* ⟩ `=` ⟨ *expr* ⟩
| `while` `(` ⟨ *expr* ⟩ `)` ⟨ *expr* ⟩
| `(` `while` `(` ⟨ *expr* ⟩ `)` ⟨ *expr* ⟩ `)` `invariant` `(` ⟨ *expr* ⟩ `)`
| ⟨ *lambda* ⟩
⟨ *type_ascript* ⟩ ::= `:` ⟨ *type* ⟩
⟨ *case* ⟩ ::= `case` ⟨ *pattern* ⟩ ⟨ *pattern_guard* ⟩? `=>` ⟨ *expr* ⟩
⟨ *pattern_list* ⟩ ::= ⟨ *pattern* ⟩ | ⟨ *pattern* ⟩`,`⟨ *pattern_list* ⟩
⟨ *pattern* ⟩ ::= ⟨ *binder* ⟩ | ⟨ *binder* ⟩ ⟨ *type_ascript* ⟩
| ⟨ *binder* ⟩ `@` ⟨ *identifier* ⟩ `(` ⟨ *pattern_list* ⟩?`)`
| ⟨ *binder* ⟩ `@` `(` ⟨ *pattern* ⟩ `,` ⟨ *pattern_list* ⟩? `)`
| ⟨ *identifier* ⟩ `(` ⟨ *pattern_list* ⟩? `)`
| `(` ⟨ *pattern* ⟩ `,` ⟨ *pattern_list* ⟩ `)`
⟨ *binder* ⟩ ::= ⟨ *identifier* ⟩ | `_`
⟨ *pattern_guard* ⟩ ::= `if` ⟨ *expr* ⟩

Figure 1.2 – Grammar for the expressions of the Leon input language.

$$
\begin{array}{rcl}
\langle\, \mathit{type} \,\rangle & ::= & \texttt{Int} \mid \texttt{BigInt} \mid \texttt{Boolean} \mid \texttt{Char} \mid \texttt{Unit} \\
& \mid & \texttt{Set}\,[\,\langle\, \mathit{type} \,\rangle\,] \mid \texttt{Map}\,[\,\langle\, \mathit{type} \,\rangle\,,\,\langle\, \mathit{type} \,\rangle\,] \mid \texttt{Array}\,[\,\langle\, \mathit{type} \,\rangle\,] \\
& \mid & \langle\, \mathit{type} \,\rangle\,\texttt{=>}\,\langle\, \mathit{type} \,\rangle \\
& \mid & \langle\, \mathit{identifier} \,\rangle\,\langle\, \mathit{instantiation} \,\rangle? \\
\langle\, \mathit{instantiation} \,\rangle & ::= & [\,\langle\, \mathit{type\_list} \,\rangle\,] \\
\langle\, \mathit{type\_list} \,\rangle & ::= & \langle\, \mathit{type} \,\rangle \mid \langle\, \mathit{type} \,\rangle\,,\,\langle\, \mathit{type\_list} \,\rangle
\end{array}
$$

Figure 1.3 – Grammar for the types of the Leon input language.

manipulate them with the standard assignment statement:

```
def foo(): Int = {
  var x = 3
  x = x + 1
  x
}
```

Note that, thanks to the rich syntactic sugar of Scala, we obtain without cost many convenient expressions such as `+=`.

The main control-flow primitives for the language are conditionals and while loops. Conditional expressions are pervasive in programming, especially with pure and recursive functions. Although they could be encoded using special ASTs, they are built-in at the solver level in Leon. `while` loops are the other way to control the execution flow in Leon.

It is also worth noting that Leon respects the same evaluation order as Scala, as well as arbitrary depth of blocks; this enables us to write somewhat convoluted, but valid code, such as

```
var x = 2
{x += 1; x} + {x *= 2; x}
```

Leon primitive types include 32 bits integers (`Int`), infinite-precision integers (`BigInt`), and booleans. Arrays, maps, and sets can also be considered as primitives, and can be built from an empty constructor and insert operations. Leon supports arbitrary tuples — of at least two elements — and these can be constructed with the usual Scala syntax (`x,y,z`).

**Types** Figure 1.3 shows the syntax of types in Leon. Again, this is a strict subset of the types in Scala, and compared to the rest of the language, this is an area where Leon imposes significant limitations.

In particular, Leon does not handle subtyping. Class hierarchies cannot be defined in Leon, with the small exception of algebraic data types, but those need to follow a very strict pattern: an abstract class as parent and only one layer of children to represent the constructors. Type

parameters are by default restricted to being instantiated by non-mutable type only; they can be declared as mutable but that restricts how their values can be used, as described in Section 1.2.3. The same list of type parameters on algebraic data types hierarchies need to be shared in each node; one particular consequence is the need to define Nil as a **case class** with a type parameter instead of as a **case object**.

The Unit type is used to play well with imperative programming, as a standard return type when a function only performs side-effects. It is also necessary for compatibility with the Scala type system, notably for assignments. Functions are first-class citizens, as can be seen from the function type. Parameters can be of function type, enabling higher-order functions.

**Annotations** Leon additionally extracts annotations on functions, methods, and classes. They are used to trigger various options and modes of Leon, and it is beyond the scope of this thesis to document all of them. However, one of these annotations — @extern [Kne16, Introduction to Chapter 4] — is used in several parts of this thesis, so it warrants a description here.

Leon can model external dependencies such as network or system interaction by using the annotation @extern. The body of a function annotated with @extern will be ignored by Leon, but the signature and the contract will still be parsed and used by Leon. The implementation should still be valid Scala code. External code can make use of any feature of Scala, as well as any external library. As Leon programs are compatible with Scala, external code can actually be executed when compiled with the Scala compiler and run on the JVM.

One useful application of @extern is to force a verification against only the contract of a function, instead of using the concrete implementation. An example would be a function that returns any integer in a range:

```
def elementInRange(x: BigInt, y: BigInt) = {
  require(x <= y)
  (x + y)/2
} ensuring(res => res >= x && res <= y)
```

If all that matters to the client code is the postcondition — that elementInRange returns a number between x and y — then using @extern will ensure that Leon does not take into account the actual implementation of elementInRange. This is relevant if the client code takes advantage of some assumption on the known implementation of elementInRange, but such an assumption would fail if the implementation is swapped by a different one, such as returning the lower bound x. Proving a program to be correct against the abstract contract of elementInRange is stronger than proving its correctness against the concrete implementation.

### 1.2.2 Specifications

The syntax for specifications is already covered by the grammar, as most of the language is shared across implementation and specification. However, we discuss here the specificities of writing specifications in Leon.

User-provided specifications in Leon take the form of contracts for functions, assertions, class invariants, and loop invariants. A function contract is the combination of a precondition (**require**) and a postcondition (**ensuring**):

```
def factorial(n: BigInt): BigInt = {
  require(n >= 0)
  if(n == 0) 1 else n*factorial(n-1)
} ensuring(res => res >= 0)
```

The same **require** is used to declare class invariants in the constructors of case classes:

```
case class Rational(n: BigInt, d: BigInt) {
  require(d != 0)
}
```

Class invariants are checked at instantiation time, and whenever a field is updated. In particular, the invariant is not checked only on function entry and function exit, but also in the middle of a function. A temporary invalid state will lead to an invariant violation.

Assertions are expressed with the usual **assert** statement, can be placed anywhere in the code, and are supposed to be valid for all possible execution paths. Finally, loop invariants are declared with the **invariant** keyword wrapping a loop:

```
(while(x < N) {
  x += 1
}) invariant(x >= 0)
```

The whole functional core can be used for writing specifications. In particular, this means that specifications can use any pure functions, including recursive ones. Usage of state is not allowed in specifications. Just as it is poor style to modify the state in an **assert** statement, it is quite dangerous to have contracts perform effects on some external state. However, functions can still be pure even when using local variables and while loops; and such functions can be used in the specifications.

Although specifications are not allowed to mutate values, the specifications can still refer to mutable values, therefore we need to clearly define at what point the specifications are checked. Loop invariants are checked before entering the loop (the initial state), and after each loop iteration (the inductive step). A precondition is checked on entering the function, before any other statement has been executed. A postcondition is checked after the returned expression has been computed. A reference to one of the parameters in a postcondition will refer to the final value of the object when returning from the function.

With the introduction of mutable state, we extend the specification language with the old construct that can be used around identifier to refer to their value when entering the function. The old keyword is only supported in postconditions. It serves to specify the update performed to a value:

```
def updateX(a: A): Unit = {
  a.x += 1
} ensuring(_ => a.x == old(a).x + 1)
```

Note that the old expression refers to a deep copy of the original object. It behaves as if the entire portion of the heap used by the object had been copied and conserved to check the postcondition.

When implementing methods, old can be used to refer to the value of **this** before invoking the method. This follows the same semantics as when using old with a variable. However, Leon currently does not support using old directly with a field of the class. This is easily worked around by explicitly using **this** for field accesses and wrapping old around the **this** keyword.

**Executable Specifications** As specifications are simply Scala expressions, they are, for the most part, executable as well. Recall that our goal is to provide a complete library implementation that could be used along with the verified Leon program, and run on the standard Scala infrastructure, without a need for the Leon system.

Among the expressions that represent formulas for specifications, the single limitation is the old expression. We provide old as a library definition, but its runtime implementation throws an error because it is not executable. We could provide an execution model, but it would require transformations that are not local to the invocation of old.

The keywords **require**, **ensuring**, and **assert** are part of the Scala standard library and their default behavior is to dynamically check the conditions. This is good default behavior, but due to the impossibility of executing specifications containing old, we provide an alternative implementation in the Leon library; this implementation can be used on a file by file basis with an **import** statement:

```
import leon.lang.StaticChecks._
```

This alternative replaces the runtime check and silently ignores the condition. The name was chosen to reflect that the checks are only performed statically — when verifying with Leon.

The loop invariant construct is provided in the Leon library, with the help of implicit conversions, and its implementation does not check the specification at runtime. Class invariants also differ with Scala. When running a Leon program with Scala, class invariants are checked only when initially instantiating a class, but not on each field update.

All in all, these differences in semantics are a product of the design goal of running Leon

programs on a standard Scala stack. Leon actually has an internal interpreter and code generator, which are able, in principle, to execute code that closely match the semantics of the verification language. The limitation with providing a compatible library implementation can be worked around by first proving the property with Leon, then using StaticChecks to ignore the implementation at runtime: a safe optimization once the property was proven statically.

### 1.2.3 Typing and Aliasing Restrictions

We complement the formal grammar by describing the typing of the expressions and some restrictions on what expressions can be constructed. In this section, we focus on an intuitive description, rather than formal typing derivation rules. We will delve more in depth on the aliasing restrictions in Chapter 3. Here, we describe it with plain English and some code snippets.

**Typing** Typing largely follows Scala rules. In fact, our implementation reuses the first few phases of the standard Scala compiler, hence shares the same type checking rules. Given that Leon does not support subtypting, some well-typed Scala programs might be rejected by Leon. We also do not support a Null type, which means that all variables must always be assigned a default value.

**Mutable Types** We can partition types in two exclusive sets: the immutable types and the mutable types. Values of immutable types can never be updated, so they can be freely shared. Using its syntactic definition, we can determine the mutability of a type. We use this partition when presenting rules in Chapter 3, but we discuss them here as type parameters that are expected to be mutable need to be marked with a special syntax.

Most primitive types in Leon are immutable, and user-defined types are immutable by default, unless a field is specified as a variable. Similarly, type parameters are immutable, unless annotated with the Mutable context bound, as in the following:

```
case class Gen[T: Mutable](t: T)
```

The context bound is only a syntactic trick in order for an intuitive syntax to express the mutability requirement on a type parameter. It is extracted by Leon and stored as a property of the corresponding type variable.

We define mutable types recursively as follows. We say that a type $t$ is mutable if

- $t$ is of type Array.
- $t$ is a type parameter annotated as mutable.
- $t$ is defined by a **case class** with at least one **var** field.

- $t$ is defined by a **case class** with at least one field of mutable type.
- $t$ is Map[K,V] and either K or V is mutable.
- $t$ is Set[V] and V is mutable.
- $t$ is $(A_1, ..., A_n)$ and at least one $A_i$ is mutable.

Any other type is immutable. It is easy to check this property for any type definition in the program, hence to partition the types into either mutable and immutable.

Notice that function types are not mutable. In Scala, closures are actually mutable, as they point to an environment with potentially mutable data. But in Leon, functions cannot capture local state, hence do not need a hidden pointer to an environment. In order to achieve an approximation of a closure in Leon, we need to represent the state explicitly. We showed how to do so in Section 1.1. Functions themselves with mutable parameters are not mutable, only the parameters are.

**Restricted Aliasing** Mutable objects bring the major complication of aliasing. In a purely functional language, the notion of aliasing does not matter for correctness (it might still matter for optimizations). As object equality is essentially equivalent to a deep structural equality, any assignment to a variable can be considered as a copy operation. But with mutable objects, we need to track which references point to what objects, in order to handle object updates properly.

We follow the similar design principles behind linear typing [Wad90] and unique pointers. Instead of building a complicated, and imprecise, alias analysis, we limit the use of aliasing to some safe and easily verifiable form. We maintain the global invariant that, in any scope, there is at most one pointer for each mutable object. This invariant can be seen as some form of more restricted type-checking and is consistently checked in the front-end part of Leon — a program that makes use of aliasing is simply rejected by Leon. With this global assumption, we are able to encode effectful functions into purely functional functions and remain sound.

We explore the implications of this aliasing model. Consider the following function that operates on two `Array`s:

```
def f(a1: Array[Int], a2: Array[Int]): Unit = {
  require(a1.length > 0 && a2.length > 0)
  a1(0) = 10
  a2(0) = 20
} ensuring(_ => a1(0) != a2(0))
```

In general, the postcondition might not hold, as `a1` could point to the same array as `a2`. In our language, however, we never have two aliased references, so it is safe to assume they point to distinct area of the heap. Leon considers the above function valid and proves it. This is a

departure from Scala, where such a postcondition would be invalid without extra information. We believe that Leon semantics is more appropriate for such a case, as using aliasing when passing parameters to a function seems hardly useful and just plain dangerous.

Controlling aliasing can help by preventing cumbersome programs that share data too extensively. However, it is true that aliasing is very handy in many cases, therefore our approach is by no means a panacea and must be seen as a trade-off between expressiveness, safety, and efficiency of verification.

## 1.3 Discussion of Design Decisions

The Leon language is simpler than Scala, which is due to a mix of forced and willful decisions. The Scala programming language is a tremendous project, now supported by many organization across the world. It has a very rich language, and it is not realistic, even for an entire research group, to support it to the same extent. However, some restrictions that have been put in place by Leon are driven by real needs to make the language safer and more amenable to verification.

**Controlled State** We decided to avoid global state, in order to make all side effects visible in the signature of functions. In particular, a function that does not have any mutable types in its parameter list is necessarily pure. This simplifies reasoning about programs for the verification system and the programmer. In the case of the verification system, being able to distinguish pure functions from effectful functions leads to less dependencies and simpler formulas to solve. The techniques we present in this thesis encode side effects as additional return values for functions; a global variable would incur the cost of having to pass it around in every function.

We also found that by using implicit parameters, it was fairly easy to emulate global variables, while still having control over where they were accessible. The typical pattern is to define some data structure representing the world — the global variables:

```
case class World(var counter: BigInt)
```

Then, any function that needs access to one of the global variables can declare the world as one of its implicit parameters:

```
def foo(x: BigInt)(implicit world: World): BigInt = {
  world.counter += 1
  2*x
}
```

Of course, this introduces transitive dependencies to the world, if a function only calls `foo` without accessing variables in the world itself:

```
def bar(x: BigInt)(implicit world: World): BigInt = {
  foo(x) + foo(x+1)
}
```

But the implicit nature makes the code relatively clean. The advantages of such an approach is that dependencies on a global variable become explicit in the signature of the function, and it is possible to have pure functions with no such dependencies. It is also possible to break the world into several parts, and to organize which functions depend on which types of global data.

More generally in Leon, the side-effects of all functions are always visible in their signature. This means that, if none of the parameters are mutable, then the function is necessarily pure. Looking only at the type of a function, it is possible to conclude whether that function can have side-effects or not. This is particularly important for higher-order functions, where the actual implementation of a function is not always known at compile time, and we need to be able to assume the absence of effects on external state.

This all amounts together to a language in which the state is tightly controlled. No function can update external state without declaring it in its parameters list. Lambda functions need to have a type that reflects the environment they want to capture; then higher-order combinators need to account explicitly for these captured environment. There is a cost to pay in notation overhead and implementation planning, but it comes with gains in automated — and manual — reasoning about the program.

**Aliasing Restrictions** Leon programs maintain a single owner per mutable value in any execution scope. This restriction is a necessary condition for the correctness of the transformations from the imperative to functional programs that we present in Chapter 2 and Chapter 3. Without this invariant, we would need to apply alternative encoding methods. One alternative encoding is the use of a global heap shared as a function parameter, but this route could make invariants more challenging to write and verification could become harder.

Additionally, many uses of aliasing seem to introduce more confusion than necessary. It seems to be seldom necessary for a function to be invoked with several of its parameters pointing to shared data. The most common use case might be when one parameter would point to a mutable component of another parameter. When the two parameters are expected to share data, this can be worked around by passing only the parent pointer, and by managing the path to the component locally in the function. Guaranteeing a unique pointer at each function entry makes local reasoning about a piece of code much easier.

There is an argument, pointed out by a reviewer, that in a sufficiently complicated system, most objects are reachable from any point, hence sharing must happen. Although we do not deny it, we believe this is most often the case in languages with a heavy imperative and object-oriented emphasis. In our experience, most data we define in Leon are immutable and

combined in a functional style, with little dependencies. State is added in a careful way to the base language and must be explicitly used. We argue for using immutable modeling as often as possible, and for only falling back to mutable data when truly necessary. There are no sharing limitations on immutable types, which is further encouragement for relying on purely functional programming as much as possible. If we follow a disciplined approach to the introduction of state and dependencies, we believe it is possible to work around these limitations.

Finally, many programs that usually employ aliases for the implementation can be written in an alternative way that does not require using aliasing. Throughout this thesis, we present small snippets of code, as well as batches of test cases that fall into our subset. In Chapter 5, we present some more significant implementations in details.

**Specifications** As much as possible, we try to share the same language for specifications and implementation. In contrast to other tools [ZKR08, Lea07], we do not use annotations and a separate logic language for writing specifications, but we rely on existing concepts from the host language. Preconditions are valid expressions in the language, of Boolean type, whereas postconditions are lambda expressions, again from the language.

Being expressed in the host language, specifications can make use of most of the Leon language, hence can model problems very closely to the source implementation. Some expressive logical operators, such as bounded ∀, can be encoded using recursive function definitions. This system of using a programming language to write specifications should be natural to the developer, as it corresponds to writing assertions to be checked at runtime.

The base principle is to have executable specifications. When specifications are executable, we are able to check them at runtime, and the semantics become clear — the program is correct unless an assertion violation is thrown. However, we both slightly extend and slightly restrict the language of specifications.

Our restriction on the regular executable language is to disallow side-effects in contracts. In principle, the technique we present here should be able to handle side-effects at the level of contracts. However, it would make the exact semantics confusing, as a value could be updated while checking the contract. It would also prevent optimizations that remove statically checked contracts, as the programmer might be relying on their execution to perform side effects.

We also extend the specifications with some non-executable constructs, notably the old notation and quantifiers. The old notation is a convenient way to refer to a value as it was before the execution of a function, but it has no direct implementation; and simulating it would require rewriting functions to perform cloning before executing their body. Quantifiers are also part of Leon, but are presented in an independent work [VK16]. Although this thesis does not address quantifiers in any way, the techniques and features presented are compatible and integrated with them.

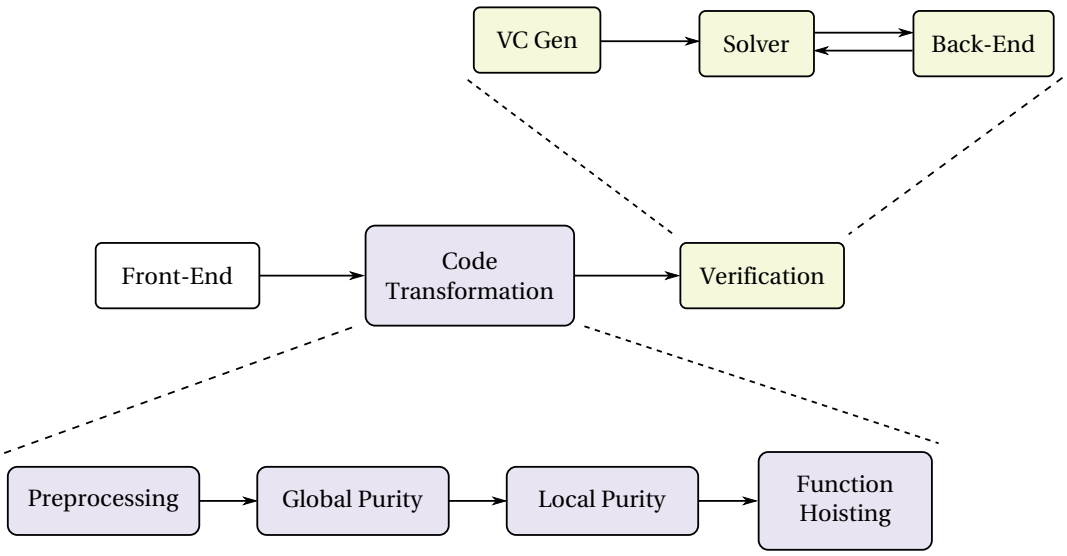


Figure 1.4 – Overall architecture of the Leon verification system.

These notations are mainly useful in the context of contracts. Although quantifiers could be used as a Boolean expression in a regular program, they lack a concrete execution plan and become essentially a constraint-solving problem. The semantics of `old` is closely related to the function to which the contract is attached, and their execution outside of contracts seems to be of little interest.

The difficulty of executing some specification notations reinforces the choice to not rely on the effects in contracts. Ideally, we would like to remove contracts after the verification step and to only evaluate the actual implementation.

## 1.4 System Architecture

We complete this chapter with an overview of the Leon verification system architecture. Leon is a joint work and parts of this architecture arose naturally in development. In order to be comprehensive, we give a quick description of each component, although our contributions are limited to the elements specifically addressed in the rest of the thesis. In this thesis, we defend a layered approach to program verification, with successive reductions of features into smaller and simpler languages. We found that this approach helps us in managing complexity in the system, by separating concerns about semantics of operations and the mapping to logical formulas.

Figure 1.4 shows the overall architecture of Leon. The input program goes through a pipeline of phases, applying several semantics-preserving transformations. At the end of the pipeline, the program is expressed in a purely functional language. This final form is handled by the solving infrastructure of Leon.

**Front-End** The front-end connects the Scala compiler to the Leon system. It uses the Scala compiler API and extracts trees after the *refchecks* phase. The trees are then translated to an internal representation used throughout the Leon system. This representation differs from the Scala representation; this allows us to evolve independently of the Scala compiler.

**Preprocessing** The initial phase — actually a series of phases — handles mostly syntactic preprocessing. It works on a syntax-tree representation of the grammar in Section 1.2. The preprocessing phase performs trivial simplifications that only require local code transformations without any particular analyses.

One simple, yet convenient, code transformation is the lifting of methods. The language we present in this thesis supports objects and methods, but without the notion of method overriding. We do not address dynamic dispatch in this thesis; and to simplify the discussion in the rest of the thesis, we see a method as a regular function. Each method $m$ defined on a class $C$ is transformed to a method $m'$ with an extra parameter (in leading position) of type $C$. Any reference to **this** within the body of the method is replaced by a reference to the fresh parameter. At each call site, $x.m(e_1, \ldots, e_n)$ is rewritten to $m'(x, e_1, \ldots, e_n)$. In the rest of the thesis, we will only discuss function invocations.

**Global Purity** The next phase is the focus of Chapter 3. It performs a global transformation on the program and rewrites all functions to a store-passing style. The output language no longer contains objects with state; and all functions are pure.

**Local Purity** This phase modularly transforms each function, removing local state. We cover the algorithms in Chapter 2. The transformation does not modify the signature of functions, maintaining the top level structure of the program.

**Function Hoisting** A final flattening is performed on the program, by hoisting nested functions to the top level. This transformation is purely syntactic, as state has been eliminated earlier in the pipeline. Without state, lifting only requires extending the list of parameters with variables coming from the outer scope. We capture path conditions on those values, and add them to the precondition of the local function [BKKS13].

**Verification** The output of the transformation pipeline is a higher-order purely functional program. The verification step first generates verification conditions for many properties of this program (most notably contracts of functions), and then the Leon solver checks them. The solver internally maps expressions to logical formulas and emits them to an external back-end, most often an SMT solver with support for the special theories of algebraic data types, arrays, integers, and bit-vectors. The solver specifically handles recursive functions with

an incremental unrolling, and it only requires the underlying logic to support the theory of uninterpreted functions. We discuss some aspects of the solving infrastructure in Chapter 4.

# 2 Eliminating Local State

In Section 1.4, we presented our layer-based approach for verifying a program written in a rich and expressive language. In this chapter, we present the local transformations that encode imperative code into an equivalent and purely functional code. This work was originally presented at Scala 2013 [BKKS13], with ideas originated in a master's thesis [Bla12]. We present an updated version of the transformation rules that are more complete and closer to the current implementation in Leon. We also extend the material, addressing additional program transformations, and discussing properties of our algorithm; and we update the evaluation to the latest state of the system.

Contrary to the subsequent Chapter 3, the present chapter only addresses *local* state. This phase does not modify the signatures of existing functions; they remain entirely pure. It does, however, rewrite local imperative implementations of functions to reduce them to a pure composition of expressions. This final form, without state, is easier to reason about in the solving infrastructure. As our transformation handles each function locally, it is modular and can scale well with a high number of small functions.

We start by introducing the idea of translating an imperative algorithm in a purely functional language. This section serves as an intuition for deriving formal transformation rules later in the chapter. We present rules that rewrite imperative expressions from the Leon input language and preserve the same semantics. The output program is equivalent to the original, but in a smaller and simpler language. We complete this chapter with some experiments and a discussion of the properties of our algorithm.

## 2.1 Functional Implementation of Imperative Algorithms

In the way of thinking, there is a fundamental difference between imperative and functional style. In imperative style, we tend to think at a lower level, with a series of instructions that modify some machine state and compute a result. With a functional style, the exact order of execution becomes less relevant, and we think more mathematically. The computation

is defined by the dependencies between data — we first compute intermediate results then combine them together into a final result.

Neither of these styles is intrinsically superior. Functional programming is often extremely concise and direct, as it does not require data to be moved around and updated. Nevertheless, some algorithms are more difficult to express in a purely functional way, especially when they rely on a large dimension of intermediate values.

Take an example, originally presented by the creator of Scala Martin Odersky[1]. We are given a list of items and want to compute the total of both the price and the discount in a single pass through the data. The canonical functional way of doing so is

```
val (totalPrice, totalDiscount) =
  items.foldLeft((0.0, 0.0)) {
    case ((tprice, tdiscount), item) =>
      (tprice + item.price, tdiscount + item.discount)
  }
```

It uses foldLeft as a way to iterate over the data and maintain the two variables that need to be computed. Now look at the canonical imperative solution:

```
var totalPrice, totalDiscount = 0.0
for(item <- items) {
  totalPrice += item.price
  totalDiscout += item.discount
}
```

It is arguably much clearer, and is undeniably shorter.

The above example also serves to illustrate a systematic way of rewriting imperative algorithms into functional algorithms. Folding is done in order to compute new intermediate values, given current intermediate values. In the general case of a **while** loop with an arbitrary condition, we can use a recursive function that performs looping with its recursion pattern.

The core idea is to take the state part and make it into an explicit value. This value can be immutable and represents the current state at the given program point. An assignment instruction then should be seen as a function that takes the current state and returns the updated state. In other words, the state is immutable, and the program performs copies for each instruction that should mutate it. In its simplest form, this is similar to static single assignment (SSA) form [AWZ88, CFR+89], which is a form of functional programming [App98]. Consider a small snippet with two local variables, x and y:

[1] Martin Odersky's Keynote at ScalaDays 2013

```
var x = 0
var y = 4
x = x + 1
y = y + x
x = y + 2
```

For each instruction, the state is the current mapping of values for x and y. We can use immutable intermediate values and consider each instruction as a mapping from the current state to the new state:

```
val x1 = 0
val y1 = 4

//x = x + 1
val x2 = x1 + 1
val y2 = y1

//y = y + x
val x3 = x2
val y3 = y2 + x2

//x = y + 2
val x4 = y3 + 2
val y4 = y3
```

Here we were quite systematic about constantly building the new state after each instruction and, in particular, we can see that each instruction requires an actual copy operation for one of the two variables, and only one of them becomes modified. A more concise transformation would give us

```
val x1 = 0
val y1 = 4
val x2 = x1 + 1
val y2 = y1 + x2
val x3 = y2 + 2
```

But it requires us to be more careful about what index to use with each occurrence of the original variable. That structure is also very close to the original imperative form, which makes it easier to read and debug.

The same idea applies when instructions are part of a loop, but the transformation needs to be adapted. As instructions are no longer static, we need a way to dynamically keep track of the intermediate values. We can use the scope of a function to maintain the current value of the state, with the parameters representing the state. The looping can be reproduced with the recursion, and each stack frame can store values at the current iteration. Consider a simple

example:

```
var i = 0
while(i < N) {
  i += 1
}
```

that can be turned into

```
def rec(i1: Int): Int = if(i1 < N) rec(i1 + 1) else i1
val i2 = rec(0)
```

The rec function replaces the simple **while** loop. At each frame, its parameter i1 represents the current value of the variable i at the corresponding iteration. The condition and the body of the loop are translated into the function body, with the assignment to i replaced by an immutable function call with the updated value of i. This is really the same step as done in the SSA example, but instead of introducing static value declarations, we use the natural scoping introduced by recursive functions. The final value of i is returned by rec, and is assigned to the local value i2 — the representative of i at the end of the loop.

## 2.2 Input Language

This transformation phase happens late in the pipeline described in Section 1.4. As a result, the input language is much simpler than the full language described in Section 1.2. In particular, the input language no longer has mutable types, and all top-level functions are pure.

Statements that mutate objects are rewritten in a phase that we discuss in Chapter 3. They have been replaced by assignments to local variables. As the language no longer contains object mutations, functions do not have side effects. Arrays follow the same restrictions as general objects, with no update statements anymore, and only functional updates.

Without stateful objects, we can ignore the issue of aliasing. Variables can freely share any immutable value, as only the pointer value itself can be updated in the current language. This makes invariants easier to track: We only need to maintain information about a variable in its definition scope; and no relationship information with other variables needs to be stored.

Consider an example of a bank account that stores a checking and a savings account, and a save function that transfers from checking to savings. A typical implementation would use side effects and mutate the original bank account:

```
case class BankAccount(var checking: BigInt, var savings: BigInt)
def save(account: BankAccount, amount: BigInt): Unit = {
  require(amount > 0 && account.checking >= amount)
  account.checking -= amount
  account.savings += amount
}
```

But this does not fit into the current restrictions of our language. However, we can transform above code in a systematic way — that will be made precise in Chapter 3 by using local variables, copy operations, and modifying the signature of the save function:

```
case class BankAccount(checking: BigInt, savings: BigInt)
def save(account: BankAccount, amount: BigInt): BankAccount = {
  require(amount > 0 && account.checking >= amount)
  var newAccount = account
  newAccount = BankAccount(newAccount.checking - amount, newAccount.savings)
  newAccount = BankAccount(newAccount.checking, newAccount.savings + amount)
  newAccount
}
```

Here, by copy, we mean the concept of creating a new object while only modifying parts of it, which does not require a copy primitive. The new version of save is actually pure because it returns an updated copy of the BankAccount class after performing the operation. We were also able to strip the **var** modifiers to the fields of BankAccount.

The role of this phase is to eliminate local imperative statements. We will focus on handling a sequence of instructions, variable declarations, assignments, conditional and match expressions, while loops, and local function definitions. The output of this phase will be an equivalent program (to the original input), but expressed in the functional core language of Leon.

## 2.3 Handling Imperative Programs by Translation

We now present formally our algorithm for rewriting imperative constructs into equivalent functional constructs.

Our procedure considers a self-contained block of instructions, for example a function body, as a whole. We generalize the description by defining how to transform individual expressions in a given context. To transform an entire function, we start from the empty context.

One element of the context is the current stateful information when an execution is being executed. In our input language, this state is an assignment of local variables to values. As values are dynamic, we can only work with them symbolically, which we do by maintaining a mapping between variable names and their current symbolic values. To avoid expressions

blowing up in size due to multiple assignments to the same variable, we introduce a fresh name for the variable after each assignment and use it the next time the variable is referred.

We can store this information with a list of names and their corresponding expressions; and a mapping from variables in scope to the most recent fresh name assigned to it. The list of names and values can be interpreted in terms of Scala semantics as a scope defined by a series of **val**. Suppose, for example, that we are given the following simple piece of imperative code:

```
x = x + 1
x = x * 2
```

We do not know the current context (what is the current value of x), but we can apply the previous method for each statement, which would result in the list (x1 -> x+1, x2 -> x1*2), and a final mapping of {x -> x2}. From the list, we can track the series of fresh names introduced for each assignment to x; and how x is renamed depends on its current name. The Scala interpretation of the two pieces of information is

```
val x1 = x + 1
val x2 = x1 * 2
x = x2
```

Notice how this code is equivalent to the previous one and can be inserted in any context where the original code was valid. The code is still imperative, but has now a normal form, with only one assignment as the final statement.

The Scala representation is useful for interpreting the mathematical representation, as well as for the basis of the generated code. The final assignment can be eliminated when leaving the definition scope of a variable, consider the complete block:

```
var x = 0
x = x + 1
x = x * 2
x
```

with the variable x being defined and its final value returned at the end. We already saw how to translate the sequence of assignments into a normal form with a final mapping from x to its current representation x2; we can substitute this representation in the returned expression, which leads to a purely functional block:

```
val x = 0
val x1 = x + 1
val x2 = x1 * 2
x2
```

The normal form with a series of definitions and a final set of assignments are useful intermediate representations to recursively combine together transformation for separate blocks of code.

More generally, we can represent any imperative program fragment as a series of definitions followed by a group of parallel assignments. These assignments rename the program variables to their new names.

This is the intuition behind the mapping from program variables to their fresh identifiers representation. When we have sequences of statements, one advantage is that we can build a recursive procedure and easily combine the results.

### 2.3.1 Example

The following program computes the floor of the square root of an integer n:

```
def sqrt(n : Int) : Int = {
  var toSub = 1
  var left = n
  while(left ≥ 0) {
    if(toSub % 2 == 1)
      left -= toSub
    toSub += 1
  }
  (toSub / 2) − 1
}
```

Our transformation starts from the innermost elements; in particular, it transforms the conditional expression into the following:

```
val left2 = if(toSub % 2 == 1) {
  val left1 = left − toSub
  left1
} else {
  left
}
left = left2
```

Then it combines this expression with the rest of the body of the loop, yielding

```
val left2 =
  if(toSub % 2 == 1) {
    val left1 = left − toSub
    left1
  } else {
    left
  }
val toSub1 = toSub + 1
left = left2
toSub = toSub1
```

The final assignments can be seen as a mapping from program identifiers to fresh identifiers. The **while** loop is then translated to a recursive function, by using a similar technique:

```
def rec(left3: Int, toSub2: Int) = if(left3 ≥ 0) {
  val left2 =
    if(toSub3 % 2 == 1) {
      val left1 = left3 − toSub2
      left1
    } else {
      left3
    }
  val toSub1 = tuSub2 + 1
  rec(left2, toSub1)
} else {
  (left3, toSub2)
}
val (left4, toSub3) = rec(left, toSub)
left = left4
toSub = toSub3
```

In this transformation, we make use of the mapping information in the body for the recursive call. A loop invariant is translated into a pre- and postcondition of the recursive function. We also substitute left and toSub in the body of the recursive function. In the final step, we combine all top level statements and substitute the new variables in the returned expression:

```
def sqrt(n : Int) : Int = {
  val toSub4 = 1
  val left5 = n
  def rec(left3: Int, toSub2: Int) = if(left3 ≥ 0) {
    val left2 =
      if(toSub3 % 2 == 1) {
        val left1 = left3 − toSub2
        left1
      } else {
        left3
      }
    val toSub1 = tuSub2 + 1
    rec(left2, toSub1)
  } else {
    (left3, toSub2)
  }
  val (left4, toSub3) = rec(left5, toSub4)
  (toSub3 / 2) − 1
}
```

$$\frac{\mathsf{e} \rightsquigarrow \langle \mathsf{r} \mid T \mid \sigma \rangle}{\mathsf{x} = \mathsf{e} \rightsquigarrow \langle () \mid T[\mathsf{val}\ \mathsf{x}_1 = \mathsf{r}; \square] \mid \sigma \oplus \{\mathsf{x} \mapsto \mathsf{x}_1\} \rangle} \qquad \frac{\mathsf{e} \rightsquigarrow \langle \mathsf{r} \mid T \mid \sigma \rangle}{\mathsf{var}\ \mathsf{x} = \mathsf{e} \rightsquigarrow \langle () \mid T[\mathsf{val}\ \mathsf{x}_1 = \mathsf{r}; \square] \mid \sigma \oplus \{\mathsf{x} \mapsto \mathsf{x}_1\} \rangle}$$

$$\frac{\mathsf{e1} \rightsquigarrow \langle \mathsf{r1} \mid T_1 \mid \sigma_1 \rangle \qquad \mathsf{e2} \rightsquigarrow \langle \mathsf{r2} \mid T_2 \mid \sigma_2 \rangle \qquad \sigma = \sigma_1 \oplus \sigma_2}{\mathsf{e1}; \mathsf{e2} \rightsquigarrow \langle \sigma(\mathsf{r2}) \mid T_1[\sigma_1(T_2)] \mid \sigma \rangle}$$

$$\frac{\mathsf{t} \rightsquigarrow \langle \mathsf{r1} \mid T_1 \mid \sigma_1 \rangle \qquad \mathsf{e} \rightsquigarrow \langle \mathsf{r2} \mid T_2 \mid \sigma_2 \rangle \qquad \mathrm{dom}(\sigma_2 \oplus \sigma_1) = \overline{\mathsf{x}}}{\mathsf{if}(\mathsf{c})\ \mathsf{t}\ \mathsf{else}\ \mathsf{e} \rightsquigarrow \langle \mathsf{r} \mid \mathsf{val}\ (\mathsf{r}, \overline{\mathsf{x}}_1) = \mathsf{if}(\mathsf{c})\ T_1[\sigma_1((\mathsf{r1}, \overline{\mathsf{x}}))]\ \mathsf{else}\ T_2[\sigma_2((\mathsf{r2}, \overline{\mathsf{x}}))]; \square \mid \{\overline{\mathsf{x}} \mapsto \overline{\mathsf{x}}_1\} \rangle}$$

$$\frac{\mathsf{e} \rightsquigarrow \langle \mathsf{r} \mid T_1 \mid \sigma_1 \rangle \qquad \sigma_1 = \{\overline{\mathsf{x}} \mapsto \overline{\mathsf{x}}_1\} \qquad \sigma_2 = \{\overline{\mathsf{x}} \mapsto \overline{\mathsf{x}}_2\} \qquad T_2 = \sigma_2(T_1)}{\mathsf{while}(\mathsf{c})\ \mathsf{e} \rightsquigarrow \langle () \mid \mathsf{def}\ \mathsf{loop}(\overline{\mathsf{x}}_2) = \{\ \mathsf{if}(\sigma_2(\mathsf{c}))\ T_2[\mathsf{loop}(\overline{\mathsf{x}}_1)]\ \mathsf{else}\ \overline{\mathsf{x}}_2\}; \mathsf{val}\ \overline{\mathsf{x}}_3 = \mathsf{loop}(\overline{\mathsf{x}}); \square \mid \{\overline{\mathsf{x}} \mapsto \overline{\mathsf{x}}_3\} \rangle}$$

$$\frac{}{\mathsf{e} \rightsquigarrow \langle \mathsf{e} \mid \square \mid \emptyset \rangle}$$

Figure 2.1 – Transformation rules to rewrite imperative statements into functional ones.

### 2.3.2 Transformation Rules

Figure 2.1 shows the formal rules for rewriting imperative code into equivalent functional code. The rules define a function $e \rightsquigarrow \langle r \mid T \mid \sigma \rangle$ that constructs from an expression $e$ a result $r$, a term constructor $T$, and a variable substitution function $\sigma$. We give rules only for the core imperative statements and briefly discuss how to adapt them to the rest of the language.

These rules are a mathematical formalization of the intuition discussed so far. We define a scope of definitions, as well as maintain a mapping from program variables to current names. Note that, each time we introduce new variables, we assume they adopt fresh names. The result term is the Scala expression that can be inserted into the scope as such, and will make a functional expression equivalent. We need it to complete a scope with pure expressions. Some expressions return the unit literal, which we represent as ().

We write term constructors as terms with one instance of a special value $\square$ (a "hole"). If $\mathsf{e}$ is an expression and $T$ a term constructor, we write $T[\mathsf{e}]$ the expression obtained by applying the constructor $T$ to $\mathsf{e}$ ("plugging the hole"). We also use this notation to apply a term constructor to another constructor; in which case, the result is a new term constructor with a hole in the inner term constructor. This is essentially a function composition, if we interpret term constructors as functions from term to term. Similarly, we apply variables substitutions to variables, variable tuples, expressions, and term constructors alike, producing as an output the kind that is passed as input.

As an illustration, if $T \equiv \Box + \mathsf{y}$, $\mathsf{e} \equiv \mathsf{x} + 1$, and $\sigma \equiv \{\mathsf{x} \mapsto \mathsf{z}\}$, then we have, for instance

$$T[e] \equiv \mathsf{x}+1+\mathsf{y} \qquad T[T] \equiv \Box+\mathsf{y}+\mathsf{y}$$
$$\sigma(e) \equiv \mathsf{z}+1 \qquad \sigma(T) \equiv \Box+\mathsf{y}$$

We denote the overriding of a substitution function by $\sigma_1 \oplus \sigma_2$. This is the union of both substitutions, but with $\sigma_2$ overriding $\sigma_1$. In other words, in case the same variable is mapped by both $\sigma_1$ and $\sigma_2$, the mapping in $\sigma_2$ overrides the one in $\sigma_1$.

Here is how to apply the rules to a top-level function. Suppose a function

$$\textsf{def}\ \mathsf{f}(\overline{\mathsf{x}}) = \mathsf{b}$$

and the derivation

$$\mathsf{b} \rightsquigarrow \langle r \mid T \mid \sigma \rangle\ ,$$

we then simply plug the result into the scope:

$$\textsf{def}\ \mathsf{f}(\overline{\mathsf{x}}) = T[\mathsf{r}]\ .$$

The last rule covers terminal expressions, expressions without children. Such expressions include all literals and variables. This is a sort of base case that simply returns itself. The rule might become easier to understand when considered in combination with the rule for sequences of expressions. That latter rule composes scopes together, and it plugs in the result of the last expression, typically a variable.

The rules for the **if** expression and **while** loop are the most complicated. The intuition behind conditionals is that each branch can update different variables, so we build an expression that returns the union domain of both branch. Each branch is built from plugging the fresh variables vectors into the respective scope. The initial element of the vector represents the result from the branch. It can be the unit literal when the branches only perform side effects, but it also handles the case where the **if** expression is the final value of a block.

The rule for the **while** loop is a bit different, as it needs to introduce a **def** in the scope. It captures all local variables that are updated, which is done with the substitution derived for the body. The rule declares new fresh variables in $\sigma_2$ for the recursive function and uses this substitution to rename variables for the body. At the end, the complete loop execution is assigned to $\overline{\mathsf{x}}_3$ — the new names for $\overline{\mathsf{x}}$.

Notice the robustness of the transformations with respect to side effects nested in expressions. Take the following code:

```
x = { x = x + 1; x * 2 }
```

Applying the rules, we first need to derive the right-hand side of the assignment. Let *e* be $x = x + 1; x * 2$, and its derivation:

$$e \rightsquigarrow \langle x1 * 2 \mid \textsf{val}\ x1 = x + 1; \square \mid \{x \mapsto x1\} \rangle\ .$$

Then we combine it with the rule for assignment:

$$x = e \rightsquigarrow \langle () \mid \textsf{val}\ x1 = x + 1; \textsf{val}\ x2 = x1 * 2; \square \mid \{x \mapsto x2\} \rangle\ .$$

We can see that the term constructor builds a scope with proper ordering of effects.

For the ease of presentation, test conditions are assumed to be pure expressions. To account for the case where the test expression has side effects, it is possible to generalize the rules for loops and conditional expressions. The generalization looks similar to how the right-hand side of assignments are handled. In our implementation, we support this more general transformation. We also omit rules for pattern matching. The implementation is somewhat cumbersome, but it is essentially a generalization of the conditional expression. Our implementation in Leon also supports the complete translation rules for pattern matching. Finally, we can support single branches `if(c) t` by rewriting them to `if(c) t else ()`.

Although the remaining expressions of the language do not need special treatment, they must still apply the rules recursively and properly combine the results. For example, function applications, which cover arithmetic operations, need to apply the rules to their arguments in proper order — from left to right. The resulting scopes must be nested correctly as well, with the leftmost as the outermost scope. Doing so will ensure that any side-effect in sub-expressions will be applied in the correct order.

### 2.3.3 Loop Invariants

We omit the translation of loop invariants from the formal rules. Each **while** can be associated with a loop invariant that is a Scala `Boolean` expression. The expression typically reads mutable variables that are in the scope hosting the loop. The invariant does not have side-effects.

We use the precondition and postcondition of the generated function to translate the invariant and to conserve its meaning. The invariant is added as a precondition to the function that is then checked at the two call sites. The initial call with initial values ensures that the loop invariant is valid when the execution enters the loop. The recursive call checks the precondition to ensure that the loop invariant is inductive — that if it holds at one step, it still holds at the following step. The postcondition is used to assume the invariant at the end of the loop. To build the postcondition, we also conjunct the invariant with the negation of the loop condition.

To formally augment the rules, consider the derivation from the **while** rule and an invariant

inv. The precondition is built as follows:

$$\sigma_2(\mathsf{inv})\,,$$

whereas the postcondition is

$$\overline{\mathsf{x}}_4 \Rightarrow \{\overline{\mathsf{x}} \mapsto \overline{\mathsf{x}}_4\}(\mathsf{inv} \wedge \mathsf{c})\,.$$

### 2.3.4 Local Functions

The rule for transforming loops generates a local function that encodes the updating of variables captured from the scope. Could we apply a similar transformation to support nested functions with side effects on local variables? The answer is yes, under some conditions.

Let us consider the case of a single nested function first. The procedure is similar to the rule for **while** loops. We can apply the rules to the body of the function, thus collecting the list of variables being assigned and their current names in the term constructor. We then extend the parameters list of the function with fresh names for each of these variables, and substitute the original variables in the term constructor. The returned value is a tuple with the original result, augmented with the updated values of the captured variables.

Additional processing is necessary for converting function invocations. We skip the details of the transformation, but each call of the transformed function (either recursively in the body itself, or in the rest of the scope) needs to be transformed into a term constructor that updates all variables captured by the function. The update can be done by extracting the $n$ variables from the tuple returned and by introducing a new name for each of them in the term constructor. We also need to pass each of the additional parameters by their current name at the program point.

We implemented this extension into Leon as well, enabling the use of local functions that capture variables. As a simple example, the following program

```
def outer(x: Int): Int = {
  var y = x
  def inner(): Unit = {
    y = y + 1
  }
  inner()
  inner()
  y
}
```

would be transformed (with some simplifications for readability) into

```
def outer(x: Int): Int = {
  val y1 = x
  def inner(y2: Int): Int = {
    y2 + 1
  }
  val y3 = inner(y1)
  val y4 = inner(y3)
  y4
}
```

We can further generalize this procedure to multiple inner functions, to mutually recursive inner functions, and also to multiple levels of nested functions. The procedure remains similar, although we need to be careful how we update the function invocations and how we track transitive side-effects among functions.

One limitation with this approach is that we cannot let the inner functions escape from their scope of definition. Consider what would happen if the outer function were to return an inner function:

```
def outer(x: Int): Int => Int = {
  var last = x
  def inner(y: Int): Int = {
    val res = last + y
    last = y
    res
  }
  inner
}
```

After transforming inner, the signature changes to

```
def inner(y: Int, last: Int): (Int, Int) = ???
```

which is incompatible with the signature of outer.

Solving this problem is not trivial. If we try to forward the new type further, we would have to globally transform each use of outer to adapt to the new signature. Besides, it is not enough to return the inner function, as call sites will need the current value of the local variable last that inner captures. Assuming we would go about doing all the proper bookkeeping, we would face an issue with higher-order functions (for example, a standard map on collection), where the function parameter has a fixed type. These higher-order functions can be used with many different instantiations of function values, therefore we cannot simply transform the signature according to one particular instance.

| Benchmark | LoC | Valid/Invalid | #VCs | Time (s) |
|---|---|---|---|---|
| Arithmetic | 81 | 4/1 | 23 | 1.78 |
| ListOperations | 144 | 8/1 | 41 | 4.22 |
| AssociativeList | 93 | 2/1 | 20 | 4.27 |
| AmortizedQueue | 105 | 8/0 | 21 | 5.79 |
| *Total* | 423 | 22/3 | 105 | 16.06 |

Figure 2.2 – Summary of evaluation results. Each benchmark is a set of operations implemented with an imperative style. We report the number of verified operations and the time.

## 2.4 Evaluation

We implement operations on some purely functional data structures with local states. The signatures of these operations remain pure, but we use loops and assignments on temporary results for the implementation. We also implement some arithmetic operations using imperative algorithms. The complete sources of our benchmarks can be found in Appendix A.2. We use Leon to prove full functional correctness of several operations over these data structures. Our data structures are based on immutable lists and maintain different representation depending on the data structure.

Our results are summarized in Figure 2.2. The benchmarks were run on a computer equipped with height CPUs running at 3.70GHz and 64.0 GB of RAM. Note that Leon does not parallelize the verification. We used Z3 version 4.3.2. The column Valid/Invalid indicates the number of valid and invalid postconditions. The column #VCs refers to additional verification conditions such as preconditions, match exhaustiveness and loop invariants. The time, in seconds, is the total amount of time necessary to verify all verification conditions, excluding initial parsing and code extraction.

## 2.5 Discussion

We complete this chapter by discussing some properties of our translations and how they relate to the alternative approaches to verification of imperative programs.

**Output Code** Our algorithm performs a code transformation that maintains the semantics of the program. The output code is valid, and it can be read and understood. We maintain a correspondence between the original and the resulting program. We achieve this by keeping similar names for variables and the same ordering of expressions.

By consistently renaming intermediate values of variables with fresh variables, we provide an explicit way to trace through the liveness of a variable. Although our transformation sometimes introduce extra intermediate variables, the size of the output is linear in the size of the input. This can be seen directly in the rules, as none of them duplicate any of the

recursively computed term constructors or results. We are very careful to always name any values that will be needed later.

In the end, the output program is of roughly the same size and structure as the input. This facilitates debugging and ensures that as little information as possible is lost. Despite loops being removed, the recursive functions that replace them are tail recursive and simply make explicit the dependencies on local variables.

Our method differs from the standard way of generating verification conditions and of then proving them. Instead of using a predicate transformer (such as weakest precondition [Dij76]) to directly generate a a formula representing the semantics of imperative statement, we encode those statements into a purely functional language semantics and leave the generation of verification to a later time. We can avoid the potential explosion in size due to the weakest precondition transformation [FS01] with our transformation that uses fresh names for each assignment, and never duplicate expressions. The exponential growth of program paths is not solved, but is pushed further down to the solver. We reduced, however, a language with imperative semantics into a purely functional semantics, and we did not lose information.

**Loop and Function Hoisting** Although our reduction does not lose information and preserves semantics, the lifting of nested functions into the open top-level scope loses implicit invariants coming from the private scope.

The function-hoisting phase comes after the local state elimination of this chapter. It hoists local functions, by capturing local-scope information and by explicitly passing all the parameters when calling the function. The purpose of this phase is to simplify the program further, and reduce it to a flat list of global functions.

Nested functions typically refer to immutable values defined locally in the outer scope, therefore the hoisting phase must capture that information and add it to the list of parameters, before lifting functions to the top-level scope. It must also capture known properties of these values, such as path conditions, and must include them as preconditions on the extended parameters. This process closes the local functions; they can then be safely lifted to the global scope.

After introducing nested functions into the global scope, the solver can handle them in a fully modular way. But the downside of modularity is that the verification will assume that the function can be called at any site, as long as the site respects the precondition. However, nested functions were originally private, and all call sites are known statically. This holds for local functions that do not escape their definition scope, hence it does not hold for closures that are returned and could be called from an unknown program point.

Users of Leon will mostly experience this scenario when using loops. Assume a simple program with a loop:

```
var i = 0
while(i < 10) {
  assert(i >= 0)
  i += 1
}
```

that our local transformations will rewrite as:

```
val i1 = 0
def loop(i2: Int): Int =
  if(i2 < 10) {
    assert(i2 >= 0)
    loop(i2+1)
  } else {
    i2
  }
loop(i1)
```

The local function loop is already fully closed (it does not capture any outside variables), so it can be analyzed in a modular way. Unfortunately, it is obvious that the assertion will fail as there is no precondition on the value of i2 when entering the function.

To avoid the spurious counter-example, we need to take into account the information from the finite number of call sites. In this example, there are two call sites: the initial call with 0 and the recursive call with i2 + 1. The recursive call makes it difficult to derive a closed formula for precondition, but the weakest precondition in that case is i2 >= 0. In the general case, this can be difficult to compute, and Leon does not infer preconditions based on call sites. It is worth noting that a loop encoding will always create this shape with a base call and a recursive call, but in general there could be nested functions with an arbitrary number of call sites (although there is always a finite number of them and they are statically known). The user can easily work around this limitation by specifying a loop invariant.

**Other Approaches** Hoare logic [Hoa69] provides a formal system for reasoning about the correctness of imperative programs. The classic approach to verification of imperative programs is to generate verification conditions using weakest precondition [Dij76]. Weakest precondition is a predicate transformer that generates a necessary condition in order for a postcondition of a program to be valid. Essentially, it assigns a semantic to the program statements and encodes the correctness of the program directly into a logic formula.

Other works build on this technique to verify imperative programs by the generation of verification conditions [NL98, GC10]. Our procedure, In contrast, maps all imperative statements to a sequence of definitions (**val** and **def**). Although our procedure is inspired by the generation of verification conditions for imperative programs, it differs in that it uses a purely functional programming language to encode the imperative statements.

Some of the existing approaches suffer from an exponential size of the verification condition as a function of the size of the program fragment. Our transformation to functional programs, followed by a later generation of verification conditions, avoids the exponential growth similarly to the work of Flanagan et al. [FS01]. We use a more direct model, without weakest preconditions, but the net result is still that the exponential growth of program paths is pushed to the underlying SMT solver, as opposed to being explored eagerly.

The Why tool [BFMP11, FP13] possesses many of the features of Leon. The original description of its interpretation of imperative programs [Fil99, Fil03] shares some aspects with our method, in particular, it uses a translation to functional representation to express the predicate transformers for each imperative statement. However, it does not generate an intermediate program in a purely functional language, rather it combines the functional transformation with the generation of verification conditions.

Dafny [Lei10] is also very similar to Leon, relying extensively on SMT solvers as well. Dafny handles the same primitive imperative operators as Leon, however it does so by translating to BoogiePL [BCD+05], an intermediate language for verification. BoobiePL differs from our target language, as it is a form of nondeterministic guarded command language with a notion of state. By contrast, our approach eliminates all stateful computations.

In contrast, our reduction to PureScala has the advantage of mapping the semantics of imperative programming to a pure and simple higher-order language. PureScala is deterministic, which makes it easy to execute, either with an interpreter or with generated code. The deterministic nature and simple pure and strict semantics of PureScala makes it easier to find counter-examples.

# 3 Pure Functions and Mutable State

In this chapter, we introduce the idea of using pure functions in a world with stateful objects. This idea fits our global algorithm, reducing one set of features to a purely functional target framework.

The transformation presented here is one phase of the Leon pipeline. The input is essentially the full language described in Chapter 1, with the basic syntactic simplifications applied — in particular, methods have been lifted. The output of this phase is a program with only pure functions. The functions can still use state locally, but their behavior is defined exactly by their input/output relationship, and they do not have implicit side-effects on arguments.

This phase modifies the signature of certain top-level functions. The change of signature is necessary and reflects the effects of the function on its argument. Consider the following signature:

```
def sort(a: Array[Int]): Unit
```

An experienced programmer can tell just by looking at this signature that the implementation should sort the array in-place. Consider this alternative declaration:

```
def sort(a: Array[Int]): Array[int]
```

Arguably, Scala programmers would expect this function to not mutate the parameter and to return a fresh (and sorted) array. The technique presented in this chapter automatically transforms such functions and their bodies, so that they always return new, fresh, and updated copies of their mutable parameters.

Changing the signature of a function obviously affects each of its call sites, so this transformation must be applied to the entire program. Following our example, some function might transitively invoke sort:

```
def min(a: Array[Int]): Int = {
  sort(a)
  a(0)
}
```

After updating the signature and an implementation of sort, this call site also needs to be updated accordingly:

```
def min(a: Array[Int]): Int = {
  val sa = sort(a)
  sa(0)
}
```

Our transformation globally rewrites all functions and each of their call sites; but it does so only when needed: if the function is already pure it will not be rewritten.

This chapter expands significantly the imperative aspects of Leon's input language. Leon's strength is in reasoning with purely functional data structures and the recursive functions defined on them. The contributions presented in Chapter 2 introduced an encoding for fundamental imperative statements. However, this encoding only supports local side effects for top-level functions, that are not observable from outside. Globally, the language is still pure, with state remaining under control in local scopes, and not visible anywhere else. With the support for parameters that store mutable data, we close an important gap with respect to the real-world use of Scala. This seemingly simple extension greatly increases our ability to model effectful computation in a semantically principled way. Scala's implicit parameters enable state-like effects to be declared elegantly, whereas Leon's translations map mutable parameters and fields into pure recursive functions.

## 3.1 The Need for State

The use of state, especially global, is sometimes regarded as dangerous and hard to reason about. Part of the problem is the loss of referential transparency, hence the need for more contextual information in order to understand a block of code. We can no longer simply look at an identifier and know its value, but we need to track it through the different possible execution paths. It is arguable that functional programming is a superior style in terms of code organization and general readability.

But state is often necessary in practice. Imperative programming is closer to the underlying architecture, and can — in some cases — lead to more efficient code. It is also sometimes more natural to express an algorithm in an imperative style. And finally, external dependencies, such as input/output, are often stateful and we need a notion of state to model them.

**Efficiency** Functional programs must use copies for any operation that updates a data structure, which can be expensive in some cases. For example, updating a node in a tree requires the copy of the root and the whole path up to the node. This means that every single node along the path from the root to the updated node must be duplicated and reallocated with new values for its children. The previous nodes will also need to be garbage collected if there are no remaining pointers to them in the system.

Under certain circumstances, a smart compiler can automatically optimize copy operations by reusing the original value instead of throwing it away [DB76, Hud86, BHY89]. But this can only happen if the part of the object that is updated is not shared with any other place in the system. Besides, detecting this can be challenging. If the compiler is not able to optimize such operations, it will likely not emit a warning, which means that the programmer might be unaware of some potentially expensive operations.

Using state explicitly gives the control back to the programmer. We can now make sure that a critical piece of code will always behave as expected.

**Design Flexibility** Objects with state can sometimes be a more natural way to represent information. Although using immutable objects is generally simple and safe, some domain logic might be better represented with mutable objects. Typical examples would be objects that represent real-world entities that have a lifecycle, by opposition to abstract concepts such as numbers.

As an illustration, consider the example of a bank that maintains a list of client accounts. Here is a simple implementation using immutable classes:

```
case class Account(id: BigInt, name: String, checking: BigInt)
case class Bank(accounts: List[Account])
```

Consider a transfer operations between two accounts

```
def transfer(from: Account, to: Account, amount: BigInt): (Account, Account)
```

Already the signature of `transfer` is a bit cumbersome, as it needs to return fresh values for both updated accounts. Updating an account in the bank is also quite awkward: the whole list of accounts should be mapped to a fresh list, with only one account modified. A more fundamental limitation is that it is impossible to locally update an account object without a pointer to the complete bank object.

On the other hand, using mutable data structures greatly simplifies the implementation. Operations, such as `transfer`, can update objects locally without the need to have an additional pointer to the aggregate object. It is sufficient to have a pointer to the modified account in order to update the data; the bank object is transparently updated without additional work.

**Modeling External Dependencies** The purely functional subset of Leon works best when applied to selected components of a Scala application. Fragments that are immutable and dependency-free can be verified by Leon if they fall into the subset. The rest of the application can then trust their correctness and use them as a safe kernel of verified functionalities. However, the core components should not depend on any other components that cannot be handled by Leon.

In order to relax this limitation, we can use notations to abstract away some components and ignore their implementations. One such notation was originally introduced into Leon in the form of the epsilon operator [Bla12, Section 4.1] that lets the programmer to state only the postcondition of an expression, without providing an implementation. Using epsilon, we can declare functions whose implementation Leon would not able to handle — typically system libraries. An example is the random function:

```
def random(n: Int): Int = {
  require(n > 0)
  epsilon((x: Int) => x >= 0 && x < n)
} ensuring((res: Int) => res >= 0 && res < n)
```

which returns an arbitrary positive number in the given bound. More complicated applications include input/output and file system dependencies.

With this method, it is in principle possible to expand the scope of verification and to have dependencies from the verified parts to unsafe components. A more recent implementation, in the form of an @extern annotation, permits for the addition of arbitrary Scala code in the body as well [Kne16, Introduction to Chapter 4], hence can be used when we want to have the ability to execute the verified program.

However, when mixed with state, both epsilon and @extern implementation suffer from a soundness issue. The underlying representation of these abstractions are uninterpreted functions with a contract attached. The functions are handled by the Leon unrolling solver, that automatically instantiates the postcondition on invocations of the function. Uninterpreted functions are treated as mathematical functions hence are assumed to be pure. Some functions, such as random, have an implicit global state that ensures independence of successive invocations. Unfortunately, this implicit state is not modeled in Leon, and Leon will prove the following theorem:

```
def wrong(n: Int): Boolean = {
  random(n) == random(n)
} holds
```

which is definitely not an expected property of random. A reasonable semantic for programs that use random number generators would be that a property is valid under any possible sequence of generated numbers. The present definition of random does not have this important property.

Using effects, we can properly model functions, such as random, and not put too much of a burden on the programmer, thanks to implicit parameters. We introduce an explicit state that is passed around globally in the program, and we maintain unique information that is different at each call of the random function. The unique information can be a global counter that monotonously increases at each invocation. This additional parameter will then be used to differentiate each call and to make sure an independent value can be chosen by the solver.

Consider an extract of the implementation of Random from the official Leon library. It uses @extern instead of epsilon, as we want to be able to execute it with the official Scala infrastructure:

```
object Random {
  case class State(var seed: BigInt)
  def nextInt(n: Int)(implicit state: State): Int = {
    require(n > 0)
    state.seed += 1
    nativeNextInt(state.seed)
  } ensuring(x ⇒ x >= 0 && x < n)
  @extern
  private def nativeNextInt(n: Int)(seed: BigInt): Int = {
    scala.util.Random.nextInt(n)
  } ensuring(res ⇒ res >= 0 && res < n) }
```

We use a State object defined locally in the Random object implementation; This State is passed as an implicit parameter to the nextInt function. Notice the two-step process, with a first function that monotonously increases the state seed value, and a second function, annotated as external, that implements the random operation. We need this separation, as marking a function as external completely disregards the implementation, and we need to take into account the update to the state when apply verification.

Now, a client can make use of the library by declaring an implicit state value in a main function, and adding implicit parameters anywhere the functionality is needed:

```
def main(): Unit = {
  implicit val state = Random.newState
  assert(foo(5) == foo(5))
}

def foo(n: Int)(implicit state: Random.State): Int = {
  require(n > 0)
  Random.nextInt(n)
}
```

Here, the assertion will catch an issue, as the same state is being shared by the successive nextInt calls, each time with a distinct value. Due to implicits, the client never needs to type the state being passed around. The final code looks very close at how it would have been written in regular Scala, with only the additional parameter list and the first state declaration. It can

also be seen as a cleaner programming method, without global state, and with dependencies on state declared in the function signature.

**Modeling OS Environment** We use a similar technique for modeling the operating system environment. The implementation is available as a standard library that mimics as much as possible a subset of the standard Scala library, thus enabling users of Leon to write interactive programs. The programs can be verified against the abstraction of the environment, and then compiled with scalac and executed as a normal Scala program.

The library provides an extended implementation of the Random object described above, a way to read from standard input and write to standard output, and a minimalistic file system interface. These libraries come with some specifications that Leon will use to prove programs. As Leon will not be able to make any assumptions on data coming from the system (other than correct type, and some bound checking for certain Random functions), it is up to the programmer to handle all possible cases. In this sense, a verified program is guaranteed to not crash due to some unexpected user input.

## 3.2 A Simple Effect Analysis for Leon

Our reduction rewrites all functions such that their effects become explicit. A straightforward transformation would be to rewrite all function signatures by only looking at the mutability of the types (and disregarding the implementation). That is to say, if a function takes a mutable type $A$ as a parameter

```
def f(a: A): Unit
```

then we would automatically rewrite it to a functional style:

```
def f(a: A): A
```

We also must update all call sites accordingly.

However, this is unnecessary, as some functions will perform only read operations on their mutable parameters. These functions and their call sites would not need to be rewritten. By avoiding unnecessary transformations, we keep the translated programs as lean and as close to the originals as possible.

We thus perform an interprocedural analysis to more precisely detect the transitive effects of each function. This enables users to write read-only functions on mutable data, without additional overhead in terms of types or conversions. Given the output from the effect analysis, the actual rewriting of each function can be done modularly. Our analysis is tailored to the design of Leon. It is simple, but sufficient for our purpose.

**Objects Formalization** For now, we will only consider effects on objects, and we will ignore arrays, maps, sets, and tuples. The rules we describe for objects can be easily generalized to tuples, and we will discuss the case of arrays at the end of the chapter. Sets and maps are mostly used for specifications, and there is an argument to forbid sets (and maps) of mutable objects.

Given the input program, we assume a global set $\mathscr{M}$ of all the mutable object types. For a type $t \in \mathscr{M}$, we denote its arity (the number of fields) as $\mathrm{a}(t)$, and the identifiers for the field $\mathrm{flds}(t) = (x_1, \ldots, x_{\mathrm{a}(t)})$, and for each identifier $x$ we denote its type as $\mathrm{tpe}(x)$. For simplicity, we assume that the type of each of the fields is mutable, but we could generalize to immutable types as well. We denote the constructor function for a type $t \in \mathscr{M}$ as $\mathrm{c}(t)$. The function takes $\mathrm{a}(t)$ expressions of the proper types and returns an object of type $t$.

For an object $o$, we use $\mathrm{flds}(\mathsf{o}) = \mathrm{flds}(\mathrm{tpe}(\mathsf{o})) = (x_1, \ldots, x_n)$. We can refer to the object pointed by a field $x_i$ as $\mathsf{o}.x_i$. When referring to paths, we use $\epsilon$ for an empty path. We write $x.\epsilon$ equivalent to $x$. We use the letter $p$ to refer to an arbitrary path: $x.p$ is meant to be a single identifier $x$ followed by an arbitrary path (possibly $\epsilon$). The letter $\mathsf{o}$ usually refers to an expression that could include a path such as $\mathsf{e}.x.y$. If we write $\mathsf{o}.p$, we mean that $\mathsf{o}$ is an arbitrary expression, and $p$ a path, potentially empty.

**Aliasing Restrictions** To state our aliasing restrictions formally, we use a heap for modeling objects in memory. Suppose a heap $\mathscr{H}$ of memory locations. Each object $o$ of type $t \in \mathscr{M}$ will be spread across the heap, with each component recursively pointing to other parts of the heap. In principle, a memory cell could contain immutable types that are stored by value, as well as a set of pointers to other objects. We will assume that for an object $o$, $\mathrm{h}(o) \in \mathscr{H}$ is the memory location that stores all necessary data for $o$, including pointers to its components. Given $\mathrm{flds}(\mathrm{tpe}(o)) = (x_1, \ldots, x_n)$, $(h)(o.x_i) \in \mathscr{H}$ is another memory location that stores the object represented by $o.x_i$.

Our first restriction is that, for each object in the system, each path of pointers reaches a distinct area of the heap. Formally, for the object $o$, $\mathrm{h}(o)$ and any memory cell $\mathrm{h}(o.p)$ reachable from a path are all distinct memory locations. In particular, there are no cycles and no diamond shapes. We denote the set of all memory locations used by an object $o$ as $H_o \subseteq \mathscr{H}$.

Let us now state the restrictions at the function level. Suppose a function with $n$ parameters $x_1, \ldots, x_n$ and (without loss of generality) respective mutable types $t_1, \ldots, t_n$, we require that each set of memory locations $H_{x_1}, \ldots, H_{x_n}$ do not share a single element. This restriction must be ensured at all call sites. We generalize this restriction to an arbitrary scope in which identifiers $x_1, \ldots, x_n$ are defined; thus, it is forbidden to assign a mutable variable to a new identifier. A final restriction is that a function cannot return an object that shares memory locations with one of its parameters.

Let us look at an example, consider the following mutable type

```
case class M(var x: Int)
```

and the following illegal function

```
def foo(m1: M, m2: M): Unit = {
  val m3 = m1
  ()
}
```

This function is illegal, because we assign a mutable object m1 to a new identifier m3, creating a new scope in which two identifiers (m1 and m3) share some heap locations. Obviously the example is very simple, and it would be possible in this case to handle the aliasing easily. But we want to focus on explaining the algorithms under the simplest possible restrictions of aliasing, and we will keep the discussion of a more flexible aliasing model for the end of the chapter. Notice also that m1 and m2 cannot share heap locations, so it would be illegal to call it as follows:

```
val m = M(42)
foo(m, m)
```

The restrictions extend to pointing to subcomponent as well. Consider another mutable class

```
case class A(m: M)
```

and another function

```
def bar(a: A, m: M): Unit
```

It is forbidden to call it as follows:

```
val a = A(M(42))
bar(a, a.m)
```

It is also forbidden to create a situation with several aliases to different parts of an object:

```
val m = M(42)
val a = A(m)
```

In this example, after assigning a, we have a situation with m pointing to a component of a, or two different identifiers sharing some heap locations. The restriction on the return value also forbids the following:

```
def baz(a: A): M = a.m
```

as it returns an object that shares the same heap cell as the component of the parameter a.

Finally, keep in mind that all these restrictions are applied only on mutable types. If a component in a mutable type is immutable, it is allowed to create multiple references to it and to share them.

**Effects Computation** The effect of a function is limited, by design, to its list of parameters. For each function $f$ with parameters $x_1, \ldots, x_n$, and respective types $t_1, \ldots, t_n$, we define the efcts($f$) $\subseteq \{x_1, \ldots, x_n\}$ as the set of parameters that are mutable and can possibly be mutated by an execution of $f$. Our algorithm that computes efcts is conservative and ignores control-flow.

We can define the function efcts inductively as, any $x_i$ such that $t_i$ is mutable and there exists an update on $x_i$ in the body. A statement that updates $x_i$ is either a field assignment or a function invocation $g$ with an effect on the parameter corresponding to $x_i$. Due to our aliasing rules, the field assignment can be an arbitrary path, but must always refer to $x_i$ as the original receiver. A function invocation with effects create an effect if one of the arguments is of the form $x_i.p$ and if the function has an effect on this argument. Detecting such effects is made easy by our aliasing rules. The absence of local aliasing means that we can focus on the uses of parameters $x_i$ and do not need to track aliases as well.

Effects can be transitive: a function might not perform a field assignment itself but might still invoke another function that does. Starting with empty effects for each function, we refine the efcts function until we reach a fixed point. As each pass can only increase the effects of each function, and the size of effects for each function is bounded by the number of parameters, the algorithm terminates.

The correctness follows by construction. The only primitive statements that can lead to a modification of a function parameters is a field assignment. Variable assignments cannot update a parameter as they are limited to the modification of variables, which are never global. The only other statement that can affect the state is a function invocation. As we perform a fixed point computation, and we keep adding to the effects of each function, we will eventually detect all transitive effects.

## 3.3 Rewriting Rules

We now present the rewriting rules to automatically rewrite functions with effects. We start by applying the effects analysis from the previous section on the whole program. In the following, we assume that the efcts function is defined based on this computation.

We first introduce a copy operation that we can use instead of a field assignment. The intuition is that, given a mutable type

```
case class M(var x: Int, var y: Int)
```

and the following update:

```
m.x = 13
```

we can replace it by the following statement:

```
m = M(13, m.y)
```

$$\frac{\mathsf{flds}(\mathsf{o}) = (x_1, \ldots, x_i, \ldots, x_n) \qquad \langle \mathsf{o}.x_i \mid p \mid \mathsf{e} \rangle \rightsquigarrow \mathsf{r}}{\langle \mathsf{o} \mid x_i.p \mid \mathsf{e} \rangle \rightsquigarrow c(\mathsf{o})(\mathsf{o}.x_1, \ldots, \mathsf{r}, \ldots, \mathsf{o}.x_n)} \qquad \frac{}{\langle o \mid \epsilon \mid \mathsf{e} \rangle \rightsquigarrow \mathsf{e}}$$

Figure 3.1 – Recursive procedure to perform a copy operation on an object.

This transformation only works locally, if the identifier m is a **var**.

We formalize this transformation in Figure 3.1. The rule describes a simple recursive procedure. We describe a relation of four elements $\langle \mathsf{o} \mid p \mid \mathsf{e} \rangle \rightsquigarrow \mathsf{o}'$: where o is the receiver object that is being updated, $p$ is a (potentially empty) path of selectors in o, e is the new value to store, and finally $\mathsf{o}'$ is the new copy operation that is equivalent to the assignment. Applied to the previous example, the relation would read $\langle \mathsf{m} \mid x \mid 13 \rangle \rightsquigarrow \mathsf{M}(13, \mathsf{m}.y)$.

The path in an object denoted by $p$ can be arbitrarily long, The recursive transformation handles any depth for the update. At each level, it preserves the current values for each field, except for the field being modified.

We use this copy transformation in our rewriting. Given the effects (that are computed by an interprocedural analysis), we can perform the rewriting modularly. For a function $f$ with parameters $x_1, \ldots, x_n$, types $t_1, \ldots, t_n$, and effects $\mathsf{efcts}(f) = (x_{k_1}, \ldots, x_{k_t})$, we introduce fresh identifiers for each effect with the substitution $\sigma = \{x_{k_1} \mapsto x'_{k_1}, \ldots, x_{k_t} \mapsto x'_{k_t}\}$. We then rewrite the body $b$ of the function $f$ as follows:

$$\mathsf{def}\ f(x_1, \ldots, x_n) = \{\ \mathsf{var}\ x'_{k_1} = x_{k_1}; \ \ldots\ ; \mathsf{var}\ x_{k_t} = x'_{k_t}; (b', x'_{k_1}, \ldots, x'_{k_t})\ \}$$

with $b'$ a rewritten version of $b$ with field assignments replaced by local assignments to the corresponding variable in $\sigma$.

We will formalize how we construct $b'$, but let us first look at an example. Using the definition of class M from above, consider the function

```
def inc(m: M): Int = {
  m.x = m.x + 1
  m.x
}
```

Applying the transformation above, we obtain

```
def inc(m: M): (Int, M) = {
  var m1 = m
  ({
    m1 = M(m1.x + 1, m1.y)
    m1.x
   }, m1)
}
```

$$\frac{M \in \mathscr{M}}{\textsf{val}\ x : M = \mathsf{e} \mapsto \textsf{var}\ x : M = \mathsf{e}} \qquad \frac{\langle \mathsf{o} \mid p \mid \mathsf{e} \rangle \rightsquigarrow \mathsf{o}'}{\mathsf{o}.p = \mathsf{e} \mapsto \sigma(\mathsf{o}) = \sigma(\mathsf{o}')} \qquad \frac{}{\mathsf{e} \mapsto \sigma(\mathsf{e})}$$

$$\frac{\mathsf{efcts}(\mathsf{f}) = (x_{k_1}, \ldots, x_{k_t}) \qquad \langle \mathsf{o}_{k_1} \mid p_{k_1} \mid \mathsf{r}(2) \rangle \rightsquigarrow \mathsf{o}'_{k_1} \qquad \ldots \qquad \langle \mathsf{o}_{k_t} \mid p_{k_t} \mid \mathsf{r}(t+1) \rangle \rightsquigarrow \mathsf{o}'_{k_t}}{\mathsf{f}(\mathsf{o}_1.p_1, \ldots, \mathsf{o}_n.p_n) \mapsto \{\ \textsf{val}\ \mathsf{r} = \sigma(\mathsf{f}(\mathsf{o}_1.p_1, \ldots, \mathsf{o}_n.p_n)); \sigma(\mathsf{o}_{k_1}) = \sigma(\mathsf{o}'_{k_1}); \ \ldots\ ; \sigma(\mathsf{o}_{k_t}) = \sigma(\mathsf{o}'_{k_t}); \mathsf{r}(1)\ \}}$$

Figure 3.2 – Local rewrite rules to replace implicit effects with explicit assignments to local variables.

Note that we embed the whole body as the first element of the tuple that is returned. The second element is the variable that refers to the updated object. This code relies on the strict order of evaluation of arguments, from left to right, which is enforced by the later stages of Leon (and in particular, by the local state transformation of Chapter 2). The value of `m1` that is returned is indeed the latest version, after the update performed in the body. The body is also substituted to refer to `m1` instead of `m`, and the field assignment is replaced by a copy assignment, according to the procedure in Figure 3.1.

We formalize the rewriting of the body in Figure 3.2. We define the rewriting function as $\mathsf{e} \mapsto \mathsf{e}'$, with $\mathsf{e}$ the original expression and $\mathsf{e}'$ the result after the rewriting. We assume that $\sigma$ is defined as described above, according to the function that contains the body we are currently rewriting. We only present rules for the expressions that need to be rewritten; the catch-all case simply substitutes the expression with the new names in $\sigma$. We apply the transformation recursively to all subexpressions. We design the rules so that they can be applied on any expression and in any context, and so that the result of the translation is always equivalent to the original expression and can be preserved at the same position. This property is not trivial to maintain in the case of the function invocation, and we are able to ensure it with our use of local side-effects.

We use the notation $t(i)$ to access the $i$th elements of a tuple, using a one-based indexing (just as regular tuples in Scala). The first translation rule is a simple syntactic translation that rewrites **val** into **var** for each mutable objects. This translation simply ensures that the other rules can use assignments. This rule serves a similar role to the introduction of **var** for each mutable parameters; but as the declarations are more local, we do not need to introduce fresh identifiers and return the new values. The rewriting for field assignment is simply a formalization of what we intuitively explained previously with examples. It combines with the copy operation of Figure 3.1.

The rule for function application is a bit more dense, but remains a formalization of our intuition. Given the effects of the function, we know that the invocation will no longer return only the result, but a tuple with each of the updated values. We capture these values with the first assignment to the tuple variable $r$. This is followed by local assignments to each of the mutated objects. These are either formal parameters of the host function, or local mutable

objects (which have been made into **var**s). We use the copy operation to derive the proper update, which also supports updates to sub-components (represented by the path $p_i$s). The translated expression returns the first element of $r$, which ensures the equivalence of both expressions.

Let us look at this last rule in action. Consider the following definitions:

```
case class A(b: B)
case class B(var x: Int)
def incValues(b1: B, b2: B): Int = {
  b1.x = b1.x + 1
  b2.x = b2.x + 1
  b1.x + b2.x
}
def add(x: Int, y: Int): Int = x + y
```

and the following local snippet:

```
val a1 = A(B(13))
val b1 = B(10)
add(incValues(a1.b, b1), a1.b.x) // 39
```

After applying the translations, the incValue becomes

```
def incValues(b1: B, b2: B): (Int, B, B) = {
  var b11 = b1
  var b22 = b2
  ({
    b11 = B(b11.x + 1)
    b22 = B(b22.x + 1)
    b11.x + b22.x
   }, b11, b22)
}
```

This translation only uses the field-assignment rule. Now let us apply the translation rules to the local snippet of code that calls incValues:

```
var a1 = A(B(13))
var b1 = B(10)
add({
  val r = incValues(a1.b, b1)
  a1 = A(r._2)
  b1 = r._3
  r._1
}, a1.b.x) // 39
```

We notice that the **val** declarations become **var** declarations, with the same right-hand side. The add call does not need to be rewritten, only the first argument does. This first argument,

originally a call to incValue, becomes a block of instruction that extracts results from the call to incValues and uses local assignments to preserve the same behavior. The block can be inlined at the same position because the last instruction returns the original result.

With this transformation, we achieved the elimination of effects. For completeness, the procedure we presented in Chapter 2 will transform this code into the following:

```
val a1 = A(B(13))
val b1 = B(10)
val r = incValues(a1.b, b1)
val a2 = A(r._2)
val b2 = r._3
add(r._1, a2.b.x) // 39
```

Notice how we flatten the block and use the correct identifiers to respect the same order of evaluation. The combination of these two transformation phases eliminates all the imperative features.

Now that we presented the core ideas of the rewriting, we explain how we handle additional features that make the system more expressive.

**Arrays** Arrays are very much a special case of objects. Instead of fields, arrays have indices that point to an arbitrary number of elements. The select operation — called apply in Scala/Leon — is similar to a field access. The update operation is the equivalent of a field assignment.

We can generalize our effect analysis to consider the array update operation as an additional effect. To do so, we also require the same aliasing restrictions on array variables. We must guarantee that each element of the array points to a different area in the heap, just as with fields of objects. We can then generalize our rules for rewriting array updates as well. The rewriting uses the updated function that returns a copy of the array but with one element modified. This is entirely analogous to the rewriting into a copy operation for general objects.

We also support arrays of arrays (and more generally, arrays of anything). There is no fundamental difference with arrays of primitive types. The property that makes the transformation seamless is the restriction of aliasing, which forces each sub-array to be entirely distinct.

**Higher-Order functions** When a function is a parameter, it is essentially abstract and we have no information on the behavior of the function. As a concrete function could have effects, we need to conservatively treat each function parameter as though it could have an effect on any of its own parameters. Given a function parameter $f$ with type $(t_1, \ldots, t_n) \Longrightarrow r$, let us set the subset $T_M = \{t_{k_1}, \ldots, t_{k_m}\} \subseteq \{t_1, \ldots, t_n\}$ of all mutable types. If $T_M$ is empty, the function is necessarily pure, and we do not modify it. If the subset is non-empty, we conservatively assume that $f$ could mutate any of its $m$ mutable parameters, and we use this information for

computing effects at the call site. We rewrite its type as $(t_1, \ldots, t_n) \Longrightarrow (r, t_{k_1}, \ldots, t_{k_m})$. We then transform all applications of $f$ in the same way that we did for regular function invocations with effects. Finally, we update each site where a function is instantiated (for example, the creation of a lambda), similarly to how we would rewrite the body of a top-level function.

As we have access to the body when a function is instantiated, we can in theory do a more precise analysis of its effects. However, to be compatible with the global rewriting of function types, we cannot do so. In fact, we are forced to choose the most conservative abstraction at locations where a function is taken as a parameter. Indeed, a function parameter could be instantiated with many different concrete functions, from many different call sites. If each of these functions has different effects, the types would be incompatible if we were to analyze each concrete implementation in a more precise way. The solution would then be to duplicate any higher-order function definitions. But this is not practical, as it could lead to an exponential explosion, with function type of $m$ mutable parameters, there are $2^m$ different subset of effects.

**Type Parameters** Leon supports type parameters for classes and functions. As a type parameter can be instantiated by any types — including mutable types — at different call sites, there should be restrictions on how a value of generic type should be handled in the implementation.

We use context bounds, as a form of annotation, to enable the user tell the tool that a generic type could be mutable. A type parameter can always be marked with the Mutable context bound:

```
case class Gen[A : Mutable](a: A)
```

The system will then ensure that the implementation correctly handles any value of this type, just as it would do for any other mutable type.

If a type parameter is not set to Mutable, then the system assumes it is immutable. In particular, the system will forbid to instantiate the type parameter with a mutable type. A type parameter annotated as potentially mutable can still be instantiated by an immutable type. This is because mutable types only impose restrictions on how values can be manipulated, but they do not introduce incompatibilities with immutable types, which are themselves not restricted.

Why do we require the programmer to annotate the mutability of a type parameter instead of determining the mutability on a per-instance basis? Indeed, it would be possible to determine if an actual instance of the type Gen is mutable or not, depending on the mutability of the concrete type C used to instantiate A. However, due to our aliasing restrictions, it would not be safe to define the following function:

```
def get[A](gen: Gen[A]): A = gen.a
```

This function returns a pointer to a component of the input, thus risks introducing an alias. This function is valid when the concrete type C is immutable, as Gen[C] is immutable; but it is

invalid when C is mutable. Such definitions are actually constraining the type parameters to being immutable, which we could have infer automatically (without relying on the Mutable context bound). We eventually decided to require an *explicit* annotation, which is a recurrent theme in our design.

**Other Approaches** The traditional approach to verification generates verification conditions by combining together formulas that express new values of variables in terms of current values. Although this approach works well in a local setting (as in Chapter 2), where it is possible to know all variables in scope statically, it presents some challenges when taken to a modular settings, where a method can potentially change any variable in the heap.

*Dynamic frames* [Kas06] propose to tackle this challenge by using special specifications to indicate which variables can be modified by a method. When the set of modified variables is known and documented by the specifications, it becomes possible to generate formulas that correctly preserve the value of other pieces of state. A central issue with this approach remains the possibility of aliasing; other works try to control aliasing by imposing restrictions on the program in the form of a notion of ownership [LM04]. With dynamic frames, the theory provides a specification language that is strong enough to express the independence of two variables. In contrast, we take an approach similar to ownership: we restrict the language. Our method is less general, but is simpler when applied to programs that respect the restrictions, as it does not need to specify the modified frame for each function. The technique of implicit dynamic frames [SJP12] addresses the burden of annotation by making the declarations of frames implicit. Preconditions must declare what fields are accessible in order for the implementation to modify them. The generation of verification conditions is then able to automatically infer an upper bound to the set of modified fields. This approach enables modular verification in the context of data abstraction, where the implementation is not available. In our approach, we rely on the exact implementation of functions, by unrolling the body during the verification phase. We thus derive the set of modified variables by analyzing the actual implementation, and do not offer a specification language for declaring sets of accessible variables. Although, we enforce a disciplined use of state by making it entirely explicit in the list of formal parameters for each function; this is somewhat related to providing a predicate to declare it in the contract. Our treatment of higher-order functions shows how we can deal with functions without implementation: by assuming each mutable parameter will be modified, thus infering an upper bound to the set of modified objects.

Separation logic [Rey02] is a formalization for handling shared mutable data structures. It extends Hoare logic with a new separation operator that asserts formulas for distinct part of the heap. This provides the means for reasoning about the correctness of mutations in the context of sharable data structures. In this chapter, we study a special case, when there is essentially no sharing, and we propose a method to verify such programs. Further work based on separation logic tries to tackle the issue of composing shape analysis for mutable data structures [CDOY11].

Dafny [Lei10] handles state by using dynamic frames, hence it does not impose aliasing restrictions. The language also features *ghost state*, which is convenient when writing specifications for modules. Although our language does not technically contain ghost variables, we can use regular fields as a way to track data for specifications. Dafny requires declaring all possible updates of methods, which can be accomplished by declaring ghost variables that represent sets of objects in the `modifies` clause. Dafny has an explicit model for the heap, as a map from references to values. This is the standard approach to handling state, but it requires sharing a global heap-variable across the generated formulas. Writing invariants on the heap is also a challenging task. This is where we take a different direction and avoid the use of heap in our encoding. We instead model updated objects on an individual basis, with direct cloning. This particular technique then requires the restrictions that we impose on aliasing.

The Why tool [BFMP11, FP13] also supports mutable data structure, but with some restrictions. Contrarily to Leon, recursive data structure cannot have mutable fields; data records are used to store mutable data. The language also enforces restrictions on aliasing that needs to be controlled statically. With these restrictions, they can use a standard weakest-precondition procedure to generate verification conditions, which is related to the encoding technique we present here, although our encoding works by mapping to a programming language instead of first-order logic.

## 3.4 Extensions

We complete this chapter with a description of extensions that can be built on top of our encoding. We have not implemented these extensions into Leon, but we have a clear idea how they would fit into the system. The extensions we describe here are ready to be integrated, but are not backed by experimentations. For some of them, the question remains opened as to the judiciousness of their inclusions into the language.

### 3.4.1 Global State

In this section, we discuss how we could encode global state into the features of the language. Our encoding could be done automatically by an additional phase in the Leon pipeline, or it could be done manually by the programmer.

Let us start with an example, consider the program in Figure 3.3. We assume the program starts when entering main (meaning that the global variables are initialized to 0). The function delayedAdd implicitly depends on the variables counter and last.

In Figure 3.4, we introduce a new type, Global, to aggregate these global variables. This program is equivalent to the one in Figure 3.3 and is a valid Leon program. The use of implicit parameters for Global helps in keeping the code of main very close to the original. Instead of having an implicit global scope, the global scope is made explicit with an object.

```
object Program {
  var counter: Int = 0
  var last: Int = 0

  def delayedAdd(x: Int): Int = {
    counter = counter + 1
    val res = last + x
    last = x
    res
  }

  def main(): Unit = {
    val sum1 = delayedAdd(5)
    assert(sum1 == 5)
    val sum2 = delayedAdd(10)
    assert(sum2 == 15)
    assert(counter == 2)
  }
}
```

Figure 3.3 – A program making use of global state.

```
object Program {
  case class Global(var counter: Int, var last: Int)

  def delayedAdd(x: Int)(implicit global: Global): Int = {
    global.counter = global.counter + 1
    val res = global.last + x
    global.last = x
    res
  }

  def main(): Unit = {
    val global = Global(counter=0, last=0)
    val sum1 = delayedAdd(5)
    assert(sum1 == 5)
    val sum2 = delayedAdd(10)
    assert(sum2 == 15)
    assert(global.counter == 2)
  }
}
```

Figure 3.4 – A program where state is explicitly passed to functions that depend on it.

Generally, we can systematically collect all variables declared in the top-level scope and declare all of them as part of a single data structure. Then, for any function that accesses one of these global variables, we add an implicit parameter of this type, just as in the example. We also add such an implicit parameter when a function calls another function that needs this global parameter.

It becomes explicit with this encoding that a reference to a single global variable will create a dependency to the whole global state. It would be possible to introduce a more fine-grained encoding (for example, one object per variable), but this current encoding reflects the fact that a global variable is truly global.

We can also handle top-level **val**s with mutable types. We need to apply the same aliasing restrictions, meaning that we cannot permit the creation of a local alias to a global mutable object. We can wrap these objects into the global container object, exactly as with variables. Due to the techniques presented in this chapter, we can handle mutable objects within other mutable objects.

### 3.4.2 Local Aliasing

Our aliasing restrictions are needed to ensure that the rewriting rules work. Let us first consider why that is the case. Consider the following simple program:

```
case class A(var x: Int)

val a1 = A(10)
val a2 = a1
a2.x = a2.x + 1
assert(a1.x == 11)
```

Applying our rewriting rules naively leads to

```
var a1 = A(10)
var a2 = a1
a2 = A(a2.x + 1)
assert(a1.x == 11) //fails
```

As we can see, the transformation lost the fact that `a2` is an alias of `a1`, and the assertion no longer holds. The root of the issue is the change of field assignments to copy operations.

This direct aliasing can be handled by maintaining a substitution: essentially the alias is always replaced by the original. This approach makes sense when the programmers uses a short name to refer to a deep component in an object. In this situation, it is helpful to support this form of local aliasing.

An application of this substitution technique, which we actually implemented in Leon, is for permitting bindings to mutable variables in pattern matching. Consider, for example, a list of

mutable objects:

```
case class A(var x: Int)

abstract class List
case class Cons(head: A, tail: List) extends List
case object Nil extends List
```

and a function that increments each element:

```
def inc(l: List): Unit = {
  l match {
    case Cons(a, as) =>
      a.x = a.x + 1
      inc(as)
    case Nil => ()
  }
}
```

In the Cons case, the identifier a is an alias to a mutable component of l, and is theoretically rejected by our system. By using a systematic substitution of aliases into their original, we can achieve the following valid code:

```
def inc(l: List): Unit = {
  l match {
    case Cons(_, _) =>
      l.head.x = l.head.x + 1
      inc(l.tail)
    case Nil => ()
  }
}
```

This code no longer aliases l and is equivalent to the original We assume that l is typed as a Cons in the first case, but we could make it explicit using a cast operation.

The basic procedure to add support for this substitution is as follows. For each variable declaration (**val**/**var**), if the right-hand side is either a variable or a path of fields of a variable, we collect a new mapping from the alias to the right-hand side. We also collect such mappings in pattern matching, thus binding variables in the pattern to the corresponding path of the receiver object. This mapping is maintained along the scope (i.e., mappings coming from a pattern binding are used only within that branch). When the right-hand side is already an alias, we follow the chain until we find the original identifier and use it in the mapping. When translating field assignments, if it is performed on an alias, we first replace the alias by the original object identifier. We also take into account this mapping for the computation of effects.

Further extensions to our aliasing system are challenging. Consider the assignment to a conditional expression:

```
def f(b: Boolean, a1: A, a2: A): Unit = {
  val a3 = if(b) a1 else a2
  a3.x = a3.x + 1
}
```

We cannot simply substitute a3 by the conditional expression. We could guard each field assignment by the condition

```
def f(b: Boolean, a1: A, a2: A): (A, A) = {
  var a11 = a1
  var a22 = a2
  if(b)
    a11 = A(a11.x + 1)
  else
    a22 = A(a22.x + 1)
  (a11, a22)
}
```

But this solution requires us to duplicate a field assignment. With nested conditions, the number of branches can quickly explode.

### 3.4.3 General Closures

A closure is a lambda expression that captures variables from the environment. Variables can either be local **var** or objects of mutable types. The type of a closure actually lies about its true effects. Suppose that a variable $x$ of type $t_x$ is in scope where a lambda function $f$ is instantiated with type $t_i \Longrightarrow t_o$. In the body of $f$, the variable $x$ is updated via a simple assignment statement. The explicit type for $f$ should actually be $(t_i, t_x) \Longrightarrow (t_o, t_x)$. Note that it does not matter if any of the types are mutable or immutable, as we are considering a variable assignment. Once $f$ is rewritten, the way it is used also needs to be updated to propagate the new value of $x$. We hit a serious complication when $f$ escapes the scope where it is defined, as now the caller also needs to be aware of the effects that $f$ has on the environment where it was created.

The problem gets out of control when we want to pass this closure as an argument to a higher-order function. Due to the local effects on its creation environment, a closure has a modified signature and is no longer compatible with a higher-order type definition. Furthermore, contrary to the effects on parameters, there is no way to know in advance how the type should be transformed, as it depends on all possible local environments where closures are created.

One solution to the above would be the introduction of a global heap. We can define the heap as a Map from labels to values; and each function would be rewritten to take the heap as an additional parameter and to also return it. Local closures would save the variables they capture in the heap, and their types would then be explicit and compatible globally.

We did not integrate this global heap solution in Leon. We fear that the complexity of the proof derived from programs with global sharing would not scale well. Our solution is to restrict the programs that Leon accepts to those we know how to transform without introducing global data such as a heap. In Leon, we support closures over local variables as long as the closure does not escape its definition scope, as explained in Section 2.3.4. A program with a closure that captures local variables, and where this closure is then returned, is rejected. We showed earlier (Section 1.1) how we can encode a stateful function, that then behaves very similarly to closures.

# 4 From Scala Types to SMT Formulas

We now examine how Leon goes from Scala expressions to a formal SMT logic. We give some background on the solving algorithm in Leon, an algorithm that uses efficient SMT solving algorithms and extends them with support for recursive functions over unbounded data types in higher-order programs. We then present several contributions related to the solving infrastructure of Leon; and how these extensions support the language presented in this thesis.

In the previous chapters we presented a number of phases from the Leon pipeline. The final output of this pipeline goes through the verification module of Leon. The verification modules can be roughly understood in two separate steps: the generation of verification conditions and proving them. A verification condition is an expression over the reduced functional subset of Leon. This language combines recursive functions, algebraic data types, and higher-order functions. There is no off-the-shelf solver for such logic, so Leon has its own implementation that builds on top of existing SMT solvers.

In this chapter, we focus on describing elements of this solving infrastructure. At its core, the solver consists of an interactive communication between a module in Leon that produces logical clauses, and an underlying SMT solver that validates these clauses. Leon slowly unfolds the program by incrementally generating clauses in order to decide the truth value of the formula. It will keep unrolling function definitions as long as necessary.

The Leon solver can be understood in terms of three main components, as summarized in Figure 4.1. The unrolling algorithm's role is to encode a Leon expression into a formula in a logic that an SMT solver can understand. This involves the challenge of mapping Leon data types to simpler logic-based ones. Then, a communication layer connects Leon with an SMT solver. This layer ensures abstraction from the technology used for SMT solving, and enables the possibility for Leon to become solver agnostic. Finally, the actual underlying SMT solver is seen as a black box and is independent from Leon. The SMT solver has a well-defined input logic and checks a formula for satisfiability with the option of returning a model, when one exists.

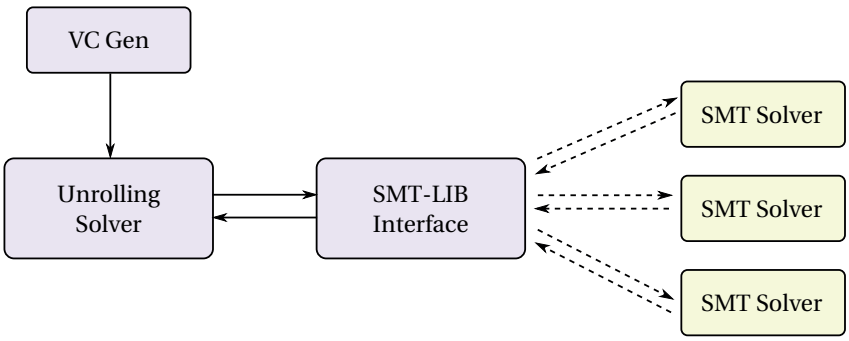


Figure 4.1 – The architecture of the solving infrastructure in Leon.

In this thesis, we make contributions to all three of these components. Although the original development of Leon was done in an independent work, the current state of this architecture is an incremental effort by many people. Here, we describe our personal contributions to various pieces in this architecture.

## 4.1 Background: Verification of Higher-Order Recursive Programs

We begin this chapter with a general background on the solver at the core of the Leon system. This solver is the result of a combined effort by many people, from the theoretical foundation [SDK10, SKK11] and the original implementation [SKK11, Sut12], to the current improved and extended version [KKKS13, Kne16]. One of the latest evolutions is the integration of higher-order functions into the theory [VKK15]. In order to support several lines of work in Leon [KKK15], additional people have contributed various implementation details to this solver.

Our specific contributions to the solver infrastructure of Leon are described in Section 4.2, Section 4.3, and Section 4.4 of this chapter. The rest of this section is a description of the unfolding algorithm that enables verification of recursive functions. Our intent is to provide the reader with a solid background for the rest of the chapter.

**PureScala** The core language in Leon is referred to as PureScala. It consists of a higher-order purely functional subset of Scala. It is the final output of the preprocessing pipeline in Leon, hence it is a subset of the language described in Chapter 1. The subset follows the same semantics as the full language, so we list only briefly what part of the language is allowed in the subset, and we refer the reader to Chapter 1 for a more detailed description of the language.

PureScala no longer contains any of the imperative features hence is purely functional. In particular, it no longer has a sequencing or assignment operator, and no local state; as such, each expression is built through compositions of pure expressions. Nested functions have been previously lifted, and methods have been encoded as functions, which makes the structure of

a PureScala program a list of top-level functions, along with algebraic data types definitions.

Types supported in PureScala include the Scala primitive types: Int, Char, and Boolean. They also include certain Scala collections native to Leon: Map, Set, and Array. And lastly, user-defined algebraic data types can be used to define custom types. Note that the arrays supported from PureScala are functional arrays and are updated with the updated method in Scala that returns a fresh copy of the array, but does not modify the array itself. Tuples are also part of the language natively.

Each top-level function can have an associated contract, consisting of either a precondition or a postcondition, or of both. The solver will prove each function's postcondition in a modular, independent way. It will also prove the validity of a precondition at each call site. But the contracts of the functions are also used while proving other properties, and they are instantiated into the logic formula each time a function call is being unrolled. Local assertions on intermediate values are also in the language.

In PureScala, functions are still first-class citizen. Functions can take other functions as parameters, and lambda expressions are part of the language.

**Verification Conditions** Once the input is reduced to PureScala, Leon generates a list of verification conditions to be checked. Each verification condition corresponds to a different correctness property of the program and will be verified in a modular way. A verification condition is a PureScala expression of Boolean type, and it can contain free variables and function invocations. An assignment to the free variables, which evaluates the formula to **false**, is a counterexample.

Generally, a verification condition is a combination of a path condition and a property that should hold at that program point. The formula states that the path condition implies the property. In practice, we solve it by going to a SAT problem, which means we negate the property, build a conjunction with the path condition, and a counterexample becomes a satisfying assignment.

The fundamental role of Leon is to verify specifications provided by the user. As we saw in Chapter 1, specifications can take several forms: contracts of functions, assertions, and loop invariants. At the PureScala stage, there are no more loop invariants, only contracts and assertions.

For each function with a postcondition, Leon will generate a verification condition by using the precondition as the path condition, and by inlining the body into the postcondition as the property to prove. Assertions are handled similarly, with one condition per assertion combined with the path condition that reaches this program point. Function preconditions are used for each static call site in the program, essentially generating an assertion that the precondition of the function invoked must hold for the given context.

Additionally, Leon will automatically generate verification conditions for implicit safety properties. These include pattern-matching exhaustiveness, array access bounds, divisions by zero, and overflow checking. These conditions ensure that Leon programs are safe and will not crash at runtime. As these conditions are implicit in the source code, they do not require explicit specifications from users and are thus conveniently checked by default.

Contract checking is not fundamentally different from other properties of the code. Leon has different rules for each sort of property and will construct the verification condition in a slightly different way; but the output of the verification conditions generation is always a list of Boolean expressions to be solved independently of each other. Although functional correctness is usually about checking the contract of a function, once it is extracted as a verification condition, it makes no difference to the Leon solver whether it is a correctness postcondition of a core function, or merely an array access bound checking.

**Solving by Unrolling** In this section, we give an overview of the algorithm for solving constraints over PureScala expressions. This whole thesis is built around this algorithm to develop an infrastructure for verifying expressive programs. For the theoretical foundations, precise formalization, and for the first experiments on functional programs, we refer the reader to the originally published research [SDK10, SKK11].

The idea of the algorithm is to determine the truth value of a PureScala Boolean expression (a *formula*) through a succession of over- and underapproximations. PureScala is a Turing-complete language, hence we cannot expect this to always succeed. Our algorithm, however, has the desirable theoretical property that it always finds counterexamples to invalid formulas. It is thus a *semi-decision procedure* for PureScala expressions.

Most of the data types of PureScala programs are readily supported by state-of-the-art SMT solvers that can efficiently decide formulas over combinations of theories such as Boolean algebra, integer arithmetic, bit-vectors, and term algebras (ADTs) [DMB08, BCD+11, DdM06]. Other PureScala data types can be encoded in a relatively straightforward way, with arrays, sets, and maps that can be implemented in the underlying theory of arrays with some extra bookkeeping. Tuples can be represented with ADTs, and strings can be seen as lists of characters or mapped to the theory of strings if the solver supports it.

One of the main challenges is in handling user-defined recursive functions. SMT solvers typically support *uninterpreted function symbols*, and we use them in our procedure. Uninterpreted function symbols are a useful overapproximation of interpreted function symbols; because the SMT solver is allowed to assume *any* model for an uninterpreted function. When it reports that a constraint is unsatisfiable, it implies that, in particular, there is also no solution when the correct interpretation is assumed. Whereas, when the SMT solver produces a model for a constraint assuming uninterpreted functions, we cannot reliably conclude that a model exists for the correct interpretation. The challenge that Leon's algorithm essentially addresses is to find reliable models in this latter case.

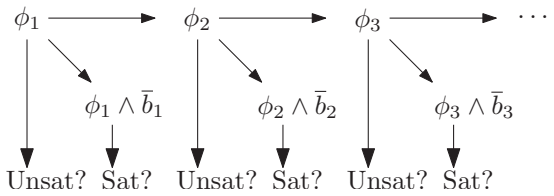


Figure 4.2 – A sequence of successive over- and underapproximations.

To be able to perform both an overapproximation and an underapproximation, we transform functional programs into logical formulas that represent partial deterministic paths in the program. For each function in a Leon program, we generate an equivalent representation as a set of clauses. For instance, for the function

```
def size(lst: List): BigInt = lst match {
  case Nil ⇒ 0
  case Cons(_, xs) ⇒ 1 + size(xs)
}
```

we produce the clauses

$$\begin{aligned} &\big(\mathsf{size}(\mathsf{lst}) = \mathsf{e}_1\big) \wedge (\mathsf{b}_1 \iff \mathsf{lst} = \mathsf{Nil}) \\ \wedge\ &(\mathsf{b}_1 \implies \mathsf{e}_1 = 0) \wedge \big(\neg \mathsf{b}_1 \implies \mathsf{e}_1 = 1 + \mathsf{size}(\mathsf{lst.tail})\big) \end{aligned} \tag{4.1}$$

Intuitively, these clauses represent the relation between the input variable lst and the result. The important difference between the two representation is the introduction of variables that represent the status of branches in the code (in this example, the variable $\mathsf{b}_1$). Explicitly naming branch variables enables us to control the parts of function definitions that the SMT solver can explore.

For example, consider a formula $\phi \equiv \mathsf{size}(\mathsf{lst}) = 1$. We can create a formula equisatisfiable —assuming the correct interpretation of size— with $\phi$, by conjoining it with the clauses (4.1). We call this new formula $\phi_1$. Now, assuming an uninterpreted function symbol for size, if $\phi_1$ is unsatisfiable, then so is $\phi$ for *any* interpretation of size. If however $\phi_1$ is satisfiable, as it is the case here, it might be because the uninterpreted term $\mathsf{size}(\mathsf{lst.tail})$ was assigned an impossible value. We control for this by checking the satisfiability of $\phi_1 \wedge \mathsf{b}_1$. This additional Boolean literal forces the solver to ignore the branch containing the uninterpreted term. If this new formula is satisfiable, then so is $\phi_1$ and we are done. If it is not, it might be because of the restricted branch. In this case, we introduce the definition of $\mathsf{size}(\mathsf{lst.tail})$ by instantiating the clauses (4.1) one more time, properly substituting lst.tail for lst, and using fresh variables for $\mathsf{b}_1$ and $\mathsf{e}_1$:

$$\begin{aligned} &\big(\mathsf{size}(\mathsf{lst.tail}) = \mathsf{e}_2\big) \wedge (\mathsf{b}_2 \iff \mathsf{lst.tail} = \mathsf{Nil}) \\ \wedge\ &(\mathsf{b}_2 \implies \mathsf{e}_2 = 0) \wedge \big(\neg \mathsf{b}_2 \implies \mathsf{e}_2 = 1 + \mathsf{size}(\mathsf{lst.tail.tail})\big) \end{aligned} \tag{4.2}$$

As $\phi_1 \wedge b_1$ was unsatisfiable, we conjoin the clauses (4.2) to $\phi_1$, obtaining $\phi_2$. We check for unsatisfiability of $\phi_2$, again assuming an uninterpreted size. It is still satisfiable, but we cannot yet know if it is due to an impossible assignment to the term size(ls.tail.tail). We next check the satisfiability of $\phi_2 \wedge b_2$, blocking the branch with the uninterpreted term. This time, it is satisfiable, as we take $e_2 = 0$ and $\neg b_1$, meaning that $e_1 = 1 + \text{size}(\text{lst.tail}) = 1$. This completes the unrolling in this example and we can soundly conclude that the formula is satisfiable.

More generally, we can repeat these steps, thus producing a sequence of alternating approximations. This process is depicted in Figure 4.2. An important property is that, although it might not necessarily derive all proofs of unsatisfiability, this technique will always find counterexamples when they exist. Intuitively, a counterexample corresponds to the property evaluating to false, and our technique enumerates all possible executions in increasing lengths.

## 4.2 Sound Reasoning about Scala Data Types

In this section, we explain the new data types that we contributed to PureScala and the Leon solver. Contrary to the extensions presented in Chapter 2 and Chapter 3, these new features require a tight integration into the solving layer. In essence, they grow the language of PureScala, offering additional types natively.

The choice to modify the design of PureScala was facilitated by the absence of direct and straightforward encodings of many of these constructs into the pre-existing language, and by the existence of similar sorts at the logic level.

First, we describe how Leon distinguishes between mathematical and bit-vector integers, and we make sure to respect the exact semantics of Scala and the JVM. This result was originally published in Scala 2015 [BK15]. Then we discuss the integration of functional arrays and their mapping to the underlying logical theory. We also list the safety conditions that we emit automatically to guarantee the proper usage of these new types, including the overflow detection feature for bit-vectors. Finally, we mention our implementation of the tuple types that are essential in transforming imperative programs into their functional equivalent forms.

### 4.2.1 Sound Integers

Previously, Leon mapped Scala's Int data type into the mathematical integers of the underlying theorem prover. We could argue that the mathematical integers correspond to many typical uses of integer type, and are appropriate for a high-level language. In languages such as Haskell, the convenient-to-use integer type `Int` denotes unbounded integers, so mapping as used in Leon up to recently would be correct.

Having access to the notion of mathematical integers indeed enables the construction of programs through well-behaved components, making programs easier to understand and

reason about. However, to model or to specify code that performs bit manipulation or code that has strong requirements on performance and memory use, a developer might want to use bounded integers, the most common choice in Scala and Java. In any case, a verification tool must provide meaningful guarantees, hence it is essential that the verification semantics conforms to the runtime semantics.

As an illustration of the semantic differences between mathematical and 32-bit integers, consider the infamous implementation of the binary search algorithm used in the java JDK[1]. This standard algorithm is present in most algorithm textbooks and was even mathematically proven correct. However, a naive proof of correctness would assume that the following statement

**val** mid = (low + high)/2

returns a number in the interval of low and high. This implicitly assumes integer semantics, as such a property does not hold with bit-vector arithmetic:

```
scala> (1000000000 + 2000000000)/2
res1: Int = -647483648
```

Because such index overflows occur only for very large data sets, it can take a very long time to detect in the field that this implementation of a binary search algorithm is not correct. A program verifier using mathematical integers to represent native integers would be fooled in the exact same way as mathematicians were believing they proved the correctness of that algorithm.

We present our modification of Leon: it distinguishes between the mathematical integers, modeled as BigInt, and the lower level concept of bit-vectors, modeled as Int. We show how reasoning about bit-vector formulas interacts with the core Leon algorithm. We discuss the treatment or the integer division, whose definition in Scala and Java differs slightly from the accepted one in mathematics, and the addition of new verification conditions for detecting divisions by zero statically. Together, these techniques give flexibility to the developers and maintain sound answers from a verification tool.

For this section, we consider a very simple functional subset of Leon and Scala, one supporting the two types Int and Boolean and a list of functions in a top level **object** definition. The functions can be mutually recursive. We build expressions with a combination of conditional expressions, standard integer, and Boolean operators.

Such a language is Turing-complete and capable enough to write interesting functions, such as

[1] http://googleresearch.blogspot.ch/2006/06/extra-extra-read-all-about-it-nearly.html

```
def factorial(n: Int): Int = {
  require(n >= 0)
  if(n == 0) 1 else n * factorial(n - 1)
} ensuring(res ⇒ res >= 0)
```

Leon encodes the above implementation into an equivalent logical formula, as a set of clauses:

$$\begin{aligned} & r = \mathsf{factorial}(n) \\ \wedge\ & n = 0 \implies r = 1 \\ \wedge\ & n \neq 0 \implies r = n \cdot \mathsf{factorial}(n-1) \\ \wedge\ & \mathsf{factorial}(n-1) \geq 0\,. \end{aligned} \tag{4.3}$$

The last clause corresponds to an inductive case: in this case Leon uses an induction hypothesis, assuming the postcondition for the recursive call. We try to prove the following property:

$$n \geq 0 \implies \mathsf{factorial}(n) \geq 0\,.$$

We can carry the proof manually with a simple case analysis. If $n = 0$, then $r = 1 \geq 0$, and if $n > 0$, then $n \neq 0$ and $r = n \cdot \mathsf{factorial}(n-1) \geq 0$ because both $n \geq 0$ and $\mathsf{factorial}(n-1) \geq 0$.

This simple proof can be done automatically in Leon, by dispatching it to an SMT solver. The SMT solver can check for satisfiability of the conjunction of clauses 4.3 with the negation of the property

$$n \geq 0 \wedge \mathsf{factorial}(n) < 0\,.$$

A model would represent a counterexample. In the present example, the conjunction is unsatisfiable as the previous proof shows.

Notice how, in the proof, the exact meaning of the function factorial does not matter. Leon actually models such functions with the theory of uninterpreted functions. The formula is still unsatisfiable, even without constraining the value of factorial. The only constraints are from the concrete unrolling steps that introduce a constraint for its value at $n-1$. This abstraction means that Leon cannot trust a counterexample to the set of clauses as a concrete counterexample to the property. A counterexample can only be trusted if it does not uses uninterpreted parts of the functions.

Unfortunately, there is a semantic gap between Scala and pure mathematics. Scala defines primitive integers as machine integers, with only a finite range; so Scala's Int is really a bit-vector of size 32. It does not take much for a function such as factorial to diverge from its mathematical definition. Whereas $13! = 6,227,020,800$, running the above implementation on 13 will give:

```
scala> factorial(13)
res0: Int = 1932053504
```

which significantly differs from the correct mathematical definition. Although the above actually verifies the postcondition of being positive, factorial(17) returns a *negative* number, violating the postcondition and throwing a runtime exception, if contracts are also checked dynamically.

This poses the question whether Leon should follow the natural mathematical meaning of the code or adhere to the exact Scala semantics. We argue for the latter. Matching Scala semantics would enable the use of Leon in real systems — those concerned with actually delivering working applications. In addition, nothing is lost because there is a Scala type, BigInt, whose semantics closely matches the one of mathematical integers. Efficiency concerns put aside, programmers should be using Int when they expect bit-vector semantics and BigInt when true mathematical integers are expected. This helps the program carry more information on its intent and gives static analysis tools a better understanding of the properties.

The proof in our example does not extend to bit-vectors. The problematic step is assuming that the product of two positive numbers is always positive. Due to overflows, this property does not translate from integers arithmetic to bit-vector arithmetic. Many important properties of integers are not verified by bit-vectors. This lack of mathematical properties complicates the task of theorem prover for a formula over bit-vectors, when compared to the same formula over integers. However, SMT solvers with the background theory of bit-vectors are still reasonably fast [JLS09, BB09, DMB08, BCD+11]. We discuss and compare the performances of the two approaches at the end of this section.

When generating verification conditions, we follow the same technique as with integers. We generate the set of clauses that correspond to the implementation of the function. Then, as before, we attempt to prove the unsatisfiability of the negation of the property. This time, however, we interpret constants and operators over the domain of bit-vectors instead of over integers. We find a concrete counterexample for factorial(17) $< 0$ and report a bug to the user.

#### Semantics of Division

There are several possible definitions for the integer division [Bou92]. In mathematics, we usually define the division of integers $x$ and $y$ as the quotient $q$ and remainder $r$ such that $q \cdot y + r = x$ and $0 \leq r < |y|$. This is known as the Euclidean definition [Bou92] and is the definition used by the SMT-LIB standard and supported by SMT solvers. The Scala programming language, following Java, defines integer division as "rounding towards zero", which differs from the Euclidean definition. In particular, the remainder — the value returned by the % operator — is sometimes negative. This definition is used both for the primitive `Int` type and the math class `BigInt`.

Leon interprets BigInt as mathematical integers. We ensure that Leon supports integer division according to Scala behavior, by encoding Scala semantics of division using the Euclidean definition. Mathematically, we define the result $q$ and denote the Euclidean division of $x$ and $y$ as $\frac{x}{y}$. A direct encoding of the Scala division as a case split is as follows:

- $x \geq 0 \wedge y > 0 \implies q = \frac{x}{y}$
- $x \geq 0 \wedge y < 0 \implies q = \frac{x}{y}$
- $x < 0 \wedge y > 0 \implies q = -\frac{-x}{y}$
- $x < 0 \wedge y < 0 \implies q = \frac{-x}{-y}$

When expressed in SMT-LIB, this encoding uses the ITE operator to do the case splitting for the different possible signs of the operands. This results in a relatively complex term with nested conditional expressions in order to express a simple division operation. The only solution for avoiding such a heavy encoding would be for the mathematical meaning of division (of SMT solvers) and the programming language meaning (of Scala) to match. For optimization, we can actually group the branches with positive $x$ and rewrite the last branch, thus we obtain the following expression for $q$:

$$\text{if } x \geq 0 \text{ then } \frac{x}{y} \text{ else } -\frac{-x}{y}\,.$$

We use the latter in our implementation, though the presence of a branching condition in the middle of an arithmetic expression is still potentially costly for the solver.

The encoding of the modulo operator is based on the result of the division operator, ensuring the correct relation between the quotient and remainder:

$$r = x - y \cdot q\,.$$

So far, we have discussed the semantics of the pure mathematical integers. The theory of bit-vectors comes with its own bvsdiv and bvsrem operators, with distinct definitions from the corresponding operators on integers. In the theory, the division is always performed as an unsigned division of the operands interpreted as positive natural numbers. The bvsdiv is provided as a short-hand notation that applies the right manipulation of number depending on their signs. The remainder is then defined to be consistent with the quotient of the division. These definitions actually match the definition of Scala for primitive Int and enables us to use a straightforward encoding of Scala division expressions into bit-vectors.

### Ensuring Safe Operations

Some operations on integers are actually unsafe. The main concerns are divisions by zero, which result in a runtime exception on the JVM. Also, though not necessarily wrong, bit-vector

arithmetic can lead to overflow, which is rarely a desirable effect.

We added support to Leon to prevent division by zero. For each division expression over integers, or bit-vector, Leon verifies that the divisor is never zero in any possible execution of the program. Leon processes such checks in the same way it handles assertions and preconditions at call sites. We build a formula that expresses the current path conditions on free variables and that the divisor expression should not be equal to zero. The verification condition is then sent to the Leon solver and proved in the same way as any other conditions would be. In particular, the expression can rely on the full language, thus Leon is able to derive safety of non-trivial divisions.

Bit-vector arithmetic overflow occurs when the value computed by a bit-vector arithmetic operation differs from the value that would be computed if both the operands and operator were to be interpreted in the integer theory. Division can lead to an overflow in a particular case (which we discuss in then end) but most issues arise with addition, subtraction, and multiplication. We derive precise bounds assuming 32-bit size, but all our rules can be generalized to any number of bits. In the following, we will use regular arithmetic operations $(+, -, \cdot)$ to represent an operation over the integers, and rounded arithmetic operations $(\oplus, \ominus, \otimes)$ for the corresponding operation over bit-vectors.

Addition and subtraction have the property of being able to overflow at most once during one computation. More precisely, if we add together two positive numbers $a$ and $b$, with $0 \leq a < n$ and $0 \leq b < n$, we get $0 \leq a + b < 2 \cdot n - 1$. With 32 bits integers, we take $n = 2^{31}$ so that $a$ and $b$ are arbitrary positive bit-vector values. We have $0 \leq a + b < 2^{32} - 1$. There is an overflow if the result is in the range $[n, 2 \cdot n - 2]$, which is $[2^{31}, 2^{32} - 2]$, which are all bit-vectors representation of negative numbers.

The development is similar when adding together two negative values. Suppose $-n \leq a < 0$ and $-n \leq b < 0$, then $-2n \leq a + b < 0$. Overflow occurs when $a + b \in [-2n, -n - 1]$; by taking the modulo we get the overflow range $[0, 2^{31} - 1]$. The final case of $-n \leq a < 0$, and $0 \leq b < n$, is simple as $-n \leq a + b < n$ and no overflow can occur. The symmetric case follows by commutativity (which holds on bit-vector addition).

When performing addition, one common property of overflow is that, when it occurs, the result is always of the opposite sign of the operands. In fact, the above bounds let us derive a concise check. Assume the binary representation of a number $n$ is $n = n_{32} n_{31} \cdots n_1$, no overflow occurs exactly when the following holds:

$$a_{32} = b_{32} \implies a_{32} = (a \oplus b)_{32} .$$

The check for detecting overflow of subtraction is derived similarly. We need to argue why a similar check can be used, as the usual rewriting of $a \ominus b$ to $a \oplus (\ominus b)$ suffers from the caveat that $\ominus b = b$ when $b = -2^{31}$. This identify holds if we omit the case $b = -2^{31}$. If $a$ and $b$ are of

| *Condition* | *Operation* |
|---|---|
| $a_{32} = b_{32} \implies a_{32} = (a \oplus b)_{32}$ | $a \oplus b$ |
| $a_{32} \neq b_{32} \implies a_{32} = (a \ominus b)_{32}$ | $a \ominus b$ |
| $a = 0 \vee \frac{a \otimes b}{a} = b$ | $a \otimes b$ |
| $a \neq -2^{31} \vee b \neq -1$ | $\frac{a}{b}$ |

Figure 4.3 – Conditions that ensure no overflow at runtime for the corresponding bit-vector operations.

the same sign, then their subtraction cannot overflow, but when the signs differ, we negate the sign of $b$ to rewrite the subtraction as an addition of operands with the same signs, and we use the rule from addition that the result sign should match the signs of the operands, in this case, the sign of $a$:

$$a_{32} \neq b_{32} \implies a_{32} = (a \ominus b)_{32}$$

if $b = -2^{31}$ and $a \geq 0$, notice that overflow should happen for every single possible value of $a$, as adding $2^{31}$ will wrap around into the negative numbers. When $a = 2^{31} - 1$, then $a - b = 2^{31} - 1 - 2^{31} = -1$, which is still negative. Hence, checking that the sign of the result differs from the sign of $a$ is enough to detect overflow of subtraction, and we can apply the above condition. If $a < 0$, no overflow can occur.

We now look at the multiplication. The issue with multiplication is that an overflow can occur, but the result might still be of the same sign as the two operands. Even worse, the result might be bigger than the two operands while still overflowing. We fall back on emitting an explicit check, as follows:

$$a = 0 \vee \frac{a \otimes b}{a} = b$$

The solver cannot simplify the second expression, due to the semantics of multiplication over bit-vectors. The first expression guards against a division by zero in the second expression and is safe as multiplying by zero ensures that no overflow happens.

Finally, we examine how the division operation can lead to an overflow. An integer division of any number $a$ and $b \neq 0$ has the property that $-|a| \leq \frac{a}{b} \leq |a|$ which bounds the result and prevents overflow in almost all cases. The one exceptional case is with $a = -2^{31}$, as we have seen that $\ominus a = a$ when interpreted over 32-bit integers. If $|b| > 1$, we get $-|a| < \frac{a}{b} < |a|$, and thus no overflow can occur. We already guard against $b = 0$, and $b = 1$ is just the identity. The case with $b = -1$ leads to $\frac{a}{b} = -a = |a|$ and falls in the overflow case. The overflow condition we derived is $a = -2^{31} \wedge b = -1$, and we can guard against it with:

$$a \neq -2^{31} \vee b \neq -1\,.$$

| **Solver** | **Z3** | | **CVC4** | |
|---|---|---|---|---|
| *Bench.* | BV | Integer | BV | Integer |
| *List Ops.* | 1.167 | 1.088 | 2.025 | 2.053 |
| *Insert. Sort* | 0.851 | 0.702 | 1.215 | 0.978 |
| *Merge Sort* | 0.821 | 0.269 | N.A. | N.A. |
| *Sorted List* | 1.088 | 1.152 | 1.751 | 1.717 |
| *Red-Black Tree* | 6.254 | 3.743 | 6.755 | 6.512 |
| *Amort. Queue* | 4.477 | 3.225 | 7.011 | 6.384 |
| *AVL Tree* | 3.494 | 2.836 | 8.146 | 7.103 |

Figure 4.4 – Comparing performance of verification using bit-vectors (BV) and integers.

Figure 4.3 summarizes the safety guards we derived. We emit these expressions as assertions guarding the corresponding bit-vector operations. We treat it as an optional verification condition, as overflow is well-defined in Scala and will not lead to a crash at runtime. This is essentially a warning that something might go wrong, but is not necessarily an error.

#### Evaluations

With the changes introduced in the present chapter, previous benchmarks using `Int` as a data type are rewritten to benchmarks using `BigInt`, capturing our original intent behind those benchmarks. We also consider the original benchmarks with the true semantics of `Int`, now correctly interpreted using 32-bit integers. Certain functions used for specification (such as `size`) still use `BigInt`, which suits their purpose in specification and enables us to prove basic properties such as size is non-negative. We ran a set of experiments to evaluate the difference in verification performance between these two versions of benchmarks. A snapshot of Leon, containing all the benchmarks reported here, is available on the `submission/scala-2015-bv` branch of the official Leon repository[2].

Figure 4.4 compares the performance of bit-vectors and of mathematical integers on a few different benchmarks. The experiments were run on an Intel core i7-4820K @ 3.70GHz with 64 GB RAM. We report the average of several runs of Leon on the benchmark for each of the configurations reported. The running time is shown in seconds. Not available (N.A.) are due to CVC4 that does not support non-linear arithmetic.

The use of integers in these benchmarks is not subject to problems of overflow, hence the use of bit-vector instead of integers does not influence the correctness of these particular properties. We can see that there is some overhead with the use of bit-vectors, in particular when implementing more complex data structures. However, in sorting benchmarks, the effect of using bit-vector is less noticeable.

Our benchmarks are representative of the use of integers. The List operations verify standard

[2] https://github.com/epfl-lara/leon

| Solver | Z3 | | | | CVC4 | | | |
|---|---|---|---|---|---|---|---|---|
| *Benchmark* | V | I | U | T. (s) | V | I | U | T. (s) |
| *Bin. Search* | 0 | 1 | 0 | 0.32 | 0 | 1 | 0 | 0.11 |
| *Bit Tricks* | 24 | 0 | 3 | 0.03 | 25 | 0 | 2 | 0.08 |
| *Identities* | 4 | 1 | 0 | 4.89 | 4 | 1 | 0 | 3.68 |

Figure 4.5 – Evaluation of programs that use bit-vectors, showing the numbers of valid (V), invalid (I), and unknown (U) verification conditions and the total time for the benchmark in seconds.

operations over lists of integers. They are mostly transparent to the properties of their element and the results show, as expected, close to no difference between using bit-vectors or integers. The sorting and sorted list benchmarks rely on the comparison operators in order to insert elements. Data structure benchmarks are similar in their use of comparisons, however the more complex shapes of formulas seem to make reasoning more difficult for the bit-vector solver.

Figure 4.5 summarizes experiments that involve bit-vectors only. The results list the different kinds of verification conditions generated for each benchmark. A valid (V.) verification condition corresponds to proving a property, an invalid (I.) corresponds to finding a bug, and an unknown (U.) is due to a timeout. The timeout was set to 30 seconds. The time is in seconds and is the average for solving all verification conditions that did not time out.

The binary search benchmark illustrates a typical bug that implementations of binary search can suffer from. One step of the search algorithm consists in looking up the value at the mean of the two indices. The natural implementation of the mean is $(x + y)/2$, which unfortunately can overflow when $x$ and $y$ are large. However, this is only an artifact due to the computation, as the average is always in the interval between $x$ and $y$. Leon, with support for bit-vectors, finds a counter-example on the former implementation. A correct implementation of the mean with bit-vector arithmetic is $x + (y - x)/2$. Notice that using mathematical integers, Leon does not report any counter-example, as in such case the two versions are equivalent.

We also evaluate several low-level bit-manipulation code fragments, many of them taken from the Hacker's Delight book [War02]. The operations exploit a small constant number of bit manipulations to obtain a result that we would naively solve by using a loop over all the bits. We assert the correctness by comparing the output to what the naive, loop-based, algorithm would have computed. The timeout cases could, in fact, be solved given sufficient time: in this case, about a hundred seconds.

Finally we look at a few arithmetic identities involving non-linear arithmetic. Non-linear arithmetic is undecidable over unbounded integers, whereas it is decidable but difficult over bit-vectors (indeed, it can encode the breaking of certain problems in cryptography). We use the following types of definitions to prove the validity of an arithmetic simplification:

```
def f(x: Int): Int = {
  require(x > 0 && x < 10000)
  (2*x + x*x) / x
} ensuring(res ⇒ res == x + 2)
```

Both Z3 and CVC4 are currently unable to prove this property over unbounded integers. Due to the finite domain, they manage to prove it for bit-vectors. Notice the upper bound constraint on the input: without some such upper bound, the identity would actually not hold due to an overflow. The invalid verification conditions is due to one such case.

### Further Extensions

Our results show that precise semantic modeling of integers can be more costly than the abstraction with mathematical integers. However, the overhead is often acceptable and sometimes even unnoticeable. Moreover, we demonstrated cases where bit-vector semantics was necessary in order to catch real bugs. In addition to checking division by zero, it is also possible to check for expressions that could lead to overflows and to issue a warning in such cases.

Because Scala and Java do not consider overflows of Int as an error but as well-behaved modular arithmetic data types, we explore the addition of bounded integers libraries that would automatically check for overflows. These data types would simultaneously encode developer's expectations that the integers remain small and efficient yet have mathematical properties of BigInts. Some preliminary results show that simple Scala programs written with BigInt, instead of Int, could lead to a difference in performance of two orders of magnitude. This naturally pushes developers to write code using Int, even when the intent is simply to use a mathematical integer. We believe that with the infrastructure present in Leon, we can combine the correctness of using BigInt with the efficiency of using Int via an automated optimization step.

Java and Scala additionally supports the bit-vector types Byte, Short, and Long: they are simply bit-vectors of different fixed sizes. The integration in the solver and the derivation of checks are generalized, in a straightforward way, to bit-vectors of arbitrary size. Leon currently does not support these types in its input language, but there is no fundamental limitation to their integration. We can further extend the set of operations by dealing with conversions between bit-vector types. The standard theory of bit-vectors provides the functions concat and extract; they are sufficient to define upcasting and downcasting.

The conversion between mathematical integers and bit-vectors however is more challenging. Leon currently only converts constant literals, for convenience. Whereas, variables of one type cannot be converted to the other. As of this writing, there is no standard primitive in SMT-LIB for converting between these two types, although some solvers do provide their own proprietary extensions. It is always possible to manually transform a bit-vector into a

mathematical integer, using its long expansion as a summation of digits. We doubt that such an encoding would be efficient in practice. Similarly, the conversion from a mathematical integer to a bit-vector could be done by using integer divisions instead of bit operations to extract each digit from the integer. We also need to define what occurs when an integer is too large to fit in a bit-vector of finite size. We did not experiment with these conversions in Leon, but it would definitely be an interesting route to explore.

### 4.2.2 Functional Arrays

In Chapter 3, we discussed extensively how to make array updates fully explicit. The translation relied on the availability of functional arrays, which are immutable arrays where updates are performed with cloning the complete array.

Immutable arrays are not available in Scala, but regular arrays expose an `updated` method that behaved exactly like an update on a functional array. Thus, PureScala supports the `Array` type, along with the `apply`, `updated`, and `length` methods. The full Leon language also supports the `update` method, but this operation was eliminated by the algorithms in Chapter 3.

**Representing Programming Arrays in Logic** The standard theory of arrays [Mcc62, dMB09] in logic supports the concept of a complete map from type to type. The theory definition permits the manipulation of such maps with `store` and `select` functions. The array — or map — must define a value for every single key. However, there is no built-in notion of a domain of definition for the map, which would be convenient for encoding programming arrays.

The domain of definition of Scala arrays is bit-vectors of size 32. Although accessing an array with a negative integer is bound to failure, it is a well-typed operation in Scala. In Leon, we want to protect from such unsafe accesses, so we restrict the range to the valid range, from zero to the length of the array. Also note that we use bit-vectors, and not mathematical integers, as the JVM specify the length of an array with a 32-bit integer, so does Scala and Java.

Beyond implementing `updated` and `apply`, the array also needs to be able to retrieve its length. Scala arrays are dynamic, in the sense that their length is not necessarily known at compile time, but could depend on runtime information. We need to track the length as a valid expression in Leon. Our solution is to represent an array as a tuple of two values: the raw array from the underlying logic theory and its length of a bit-vector sort with size 32. The `length` operation is then a select operation for the corresponding tuple element, whereas `updated` and `apply` can be forwarded to the underlying raw array. Tuples are further translated into the underlying logic, and we discuss their implementation in the next section.

**Safe Indexing** As discussed above, arrays cannot be safely indexed on the entire domain of 32-bit integers. Negative values, as well as values greater than the length, are invalid indices. The JVM does runtime checking of the index for each array access, and it throws a runtime

exception in cases where the value falls outside the valid range. For this reason, Scala and Java array accesses are checked operations, with some additional runtime overhead.

Leon proves statically the safety of array accesses, hence a compiler could use this information to erase runtime bound checking when the access was proved valid. Although not necessarily relevant when targeting the JVM, Leon targets alternative platforms, in particular low-level ones such as C, where that ability could be exploited.

We guard each `apply` and `updated` method with an assertion that states that the index argument is within the correct bound of zero and the length of the array (exclusive). The length of the array is available as an explicit element of the tuple-based representation. These assertions are treated similarly to those introduced for safe arithmetic in Section 4.2.1. In the end, Leon emits and proves a different verification condition for each array access.

**Global Array Invariants** Our encoding represents the length of an array as a 32-bit integer, which means the solver could potentially assign a negative value to the length. Interpreting the bit-vector as unsigned would introduce a problem with indexing and would differ from the semantics of Scala where the length of an array is of type `Int`.

One interesting trick that we could apply is to represent the length of an array with a 31-bit bit-vector, and to always interpret it as a positive value. This would ensure that any array in the models returned by the solver would be correct by construction. We would need to adapt all arithmetic operations referring to the length, by prepending an additional zero in front of the length to map it to the 32-bit integer world, but this would definitely be a feasible solution.

Ultimately, we chose to go with a different solution. We introduced the concept of invariant for a type within the solver and we ensured that any formula in which a fresh array is present would also instantiate the invariant that the length is positive for this array. We thus kept the length encoded as a 32-bit integer. But we added the invariant that the length of an array is always positive, and we instantiated when necessary this invariant whenever a new clause is unrolled in the solver.

The advantage of this solution is that it is compatible with the concept of class invariants for ADTs. Leon supports a notation for class invariants and ensures their validity at all time. However, maintaining these invariants can be tricky when we consider arrays of arrays, or arrays of ADTs with class invariants.

In our implementation, each free variable of an array type is protected by an additional clause that the length must be positive, as well as by a recursive expression that enumerates all indices from 0 up to the length and ensures that the invariants of the base type is maintained. We generate an implicit recursive function during the solving; this function iterates recursively over the whole array. We then issue the first invocation for index 0 as a new clause. A model that violates the verification condition will have to additionally respect the invariant for each

ADT and arrays. This ensures that Leon never constructs counterexamples with invalid arrays.

### 4.2.3 Tuples

Finally we discuss our implementation of tuples, as they are fundamental data types for supporting our infrastructure that rewrites imperative programs into functional programs.

In Chapter 2 and Chapter 3, we have shown how to rewrite an imperative program with state into a purely functional program that explicitly transforms the state. The tuple data type was pervasive through these encodings, as it was used as a way to describe the set of all the side effects that an expression or function could have.

There is no standard theory of tuples in logic, however, in principle they are simple aggregates of independent values and can be inlined during solving. Instead, we choose to instead encode them as underlying generic ADTs, introducing one tuple ADT for each new size needed.

We introduce new ADT definitions on-demand, only when an expression or type of the corresponding dimension is required. The ADT definition consists of a single constructor with the $n$ subtypes as $n$ parameters. The indexing operations become the selection of corresponding identifiers on the data type. By lazily introducing the definitions, we support arbitrary tuple dimensions. Although Scala tuples are limited to 22 in the input language, internal tuples introduced as part of the transformations do not have any limit in size.

## 4.3 Reusable Solver Interface

In theory, Leon could directly integrate with a specific SMT solver API. While generating clauses, Leon could send them right away to the underlying SMT solver. In fact, the original implementation of Leon did precisely this, connecting with the native bindings of Z3 [DMB08], and it is still available as one of the modes of Leon. This integration was made possible by a Scala library[3] [KKS11] that wraps the C API of the Z3 SMT solver.

There are some disadvantages to this approach. First, the integration with the chosen solver becomes extremely tight and is difficult to change. Although all SMT solvers essentially solve the same problem, their exact sets of features and how they choose to export them varies. By binding directly to the software API of Z3, Leon would essentially depend entirely on Z3 and its further developments. Second, there is an engineering issue with the complexity exported by a complete API for a system such as Z3, of which Leon only uses a small subset. It is common practice in software engineering, when dealing with existing systems, to hide the complexity behind a facade that translates between the external system and the artifact we are building. As an advantage though, a custom integration with an API can lead to less overhead.

With the rise of competing and complementing solvers to Z3, in particular CVC4 [BCD+11],

[3] https://github.com/epfl-lara/ScalaZ3

but also theorem provers such as Isabelle [NPW02] and Princess [RÖ8], there was a need to evolve the design of Leon towards a more flexible architecture that could support multiple alternative solvers at the same time.

In this section, we explain the design of a new interface for making Leon solver agnostic. With this interface, we were able to add support for CVC4 and are now supporting it as an official back-end of Leon, along with Z3. Being independent of the solver is particularly important as designing efficient decision procedures for theories used in programming languages is still a research topic and is evolving at a very fast pace.

**The SMT-LIB standard** Leon embraces the SMT-LIB standard for communicating with SMT solvers [BST10]. Many state-of-the-art solvers, including Z3 and CVC4, implement a robust support for this standard. SMT-LIB version 2 provides a scripting language for communicating with SMT solvers. This scripting language supports, in particular, a notion of stack of assertions that enables incremental solving if the underlying solver supports it properly.

The solving back-end of Leon is an abstraction over SMT-LIB, which essentially defines a transformation from the Leon representation of Scala programs to a first-order logic representation of programs. It performs an unrolling of recursive functions in a lazy manner, asserting more and more clauses to the solver, as explained in Section 4.1.

**A Scala Library for SMT-LIB** We developed and released an open-source Scala library, `scala-smtlib`: it provides a nearly complete support for the current 2.5 version of the standard. The library is open-source and available on GitHub[4] as a stand-alone package on which Leon depends.

`scala-smtlib` is a lightweight interface on top of the SMT-LIB standard that exposes a tree representation that mirrors the abstract grammar of the SMT-LIB language. At its core, the API offers a Parser that transforms an input stream into the tree representation, and a Printer that transforms a tree into a SMT-LIB compliant textual output. Building on this abstraction, `scala-smtlib` wraps solver processes into an interpreter for SMT-LIB scripts. This gives Scala programmers access to a type-safe interface to an SMT solver. The wrapper handles low-level communication with an external process, communicating over the textual standard input and output. The library comes with two implementations of the wrapper for Z3 and CVC4, but very little solver-specific code is required to add additional wrappers.

We refer to the online repository for more extensive documentation on the library.

**Solvers-specific Features and Swappable Solvers** In Leon, we have a general mapping from Leon trees to `scala-smtlib` trees. In principle, any solver that supports the SMT-LIB standard

[4] https://github.com/regb/scala-smtlib

could be then simply swapped as the process dependency. In practice, solvers still differ slightly on the syntax for some (non-standard) commands. They also differ in which theory they support.

We plug in solver-specific implementations for each of the areas where solvers tend to differ. These areas include, without being limited to, arrays and map operations, recursive data types, and more exotic theories such as strings and sets.

## 4.4 CafeSat: A Modern SAT Solver in Scala

In this section, we present CafeSat: a SAT solver implemented in Scala. The solver has a modern architecture, with a DPLL engine, and includes state-of-the-art optimizations.

Leon eventually encodes programming expressions into a formal logic. The preferred solution is to target an SMT solver, but there is also an alternative back-end targeting the Isabelle [Pau94, NPW02] proof assistant [HK16]. Although these systems have the advantage of maturity with many years of optimizations, they are necessarily highly complex and present some challenges in integration. In particular, SMT solvers are often implemented with C++, which further complicates the integration with Scala.

Besides the barrier of the technical communication, there is a strong separation between the development of Leon and the respective solvers. Leon has a domain of mostly functional and inductive data structures and uses abstractions to hide the lowest-level details of a program. SMT solvers should be general enough to reason about arbitrary formulas in their supported theory. They might specialize in some problem domain, but there is no guarantee that the solver will shine on the kind of formulas generated by Leon. Due to their high complexity, there is little hope for actually tying the development of Leon to the internals of such solver.

The development of CafeSat is an experiment in the direction of a closer interaction between Leon and the solver. We want to determine the feasibility of using Scala for implementing efficient solving-algorithms. Implementing a state-of-the-art solver entirely in Scala opens up the possibility for much tighter integration of Leon and the solver. It also creates an opportunity to specialize the solver for the kind of problems generated by Leon. A more direct integration leads to more reliability, with a better understanding of the different components. It is our hope that pursuing this endeavor can help future verifiability.

We believe CafeSat could also have applications in the broader Scala world. The current release of the Scala compiler integrates a small SAT solver for the pattern matching engine. It could benefit from a self-contained and efficient solver written entirely in Scala in order to avoid external dependencies. Complex systems on the JVM such as Eclipse also began including SAT solving technology for their dependency management engines [LBR09].

Finally CafeSat, beside being a practical tool, is also an experiment in writing high-performance software in Scala. Our goal it to prove — or disprove — that Scala is suitable for writing pro-

grams that are usually built in C++. The initial results reported here show that it is necessary to sacrifice some of the advanced features of Scala in order to attain acceptable performance. In particular, we show significant performance improvement that we gained over time, at the cost of removing some abstractions.

These results were originally reported at Scala 2013 [Bla13]. At the time of writing this thesis, there is a recent independent effort to integrate Princess [RÖ8], a Scala-based theorem prover for first-order logic modulo linear integer arithmetic, as another alternative back-end for Leon.

### 4.4.1 SAT Solving in Scala

The Boolean satisfiability problem (SAT) is one of the most important problems in computer science. From a theoretical point of view, it is the first NP-complete problem. On the practical side, it is used as a target low-level encoding for many applications. As SAT solvers are well understood and have been engineered over many years, applications often choose to rely on them rather than developing a custom solver for the domain. Often these SAT solvers are also an important building block in the more general problem of constraint solving and, in particular, as a basis for SMT solvers [GHN+04].

In the Boolean satisfiability problem, we are given a set of clauses, where each clause is a set of literals. A literal is either a propositional variable or the negation of a propositional variable. The goal is to find an assignment for the variables such that for each clause, at least one of the literal evaluates to true. This representation is called Conjunctive Normal Form (CNF).

CafeSat is a complete SAT solver implemented in Scala. It is strongly inspired by MiniSat [ES03]. CafeSat implements many recent techniques present in modern SAT solvers. CafeSat is built around the DPLL scheme [DLL62]. Boolean constraint propagation is implemented using the two-watched literal scheme introduced by Chaff [MMZ+01]. The branching heuristics is VSIDS, also introduced by Chaff. A key component of modern SAT solver is the conflict-driven clause learning [SS96, ZMMM01], allowing for long backtracking and restarting. CafeSat supports an efficient conflict analysis, with the 1UIP learning scheme and a clause minimization inspired from MiniSat.

Additionally, CafeSat exports an API for Scala. This enables some form of constraint programming in Scala, as already promoted by ScalaZ3 [KKS11]. We illustrate its ease of use and expressive power in Figure 4.6. The code implements a Sudoku solver. A Sudoku input is represented by a matrix of `Option[Int]`. We then generate nine variables for each entry, and generate all constraints required by the rules of Sudoku. The constraints state how variables from the same rows, columns and blocks of a Sudoku grid must relate to each other. Variables and constraints can be naturally manipulated, as would any regular Boolean expression in Scala.

Our library provides a new Boolean type and lifts the usual Boolean operations of Scala to enable a natural declaration of constraints. Any SAT problem can be build by combining

```
def solve(sudoku: Array[Array[Option[Int]]]) = {
  val vars = sudoku.map(_.map(_ ⇒ Array.fill(9)(boolVar())))
  val onePerEntry = vars.flatMap(row ⇒
                          row.map(vs ⇒ Or(vs:_*)))
  val uniqueInColumns = for(c ← 0 to 8; k ← 0 to 8;
                                r1 ← 0 to 7; r2 ← r1+1 to 8)
    yield !vars(r1)(c)(k) || !vars(r2)(c)(k)
  val uniqueInRows = for(r ← 0 to 8; k ← 0 to 8;
                             c1 ← 0 to 7; c2 ← c1+1 to 8)
    yield !vars(r)(c1)(k) || !vars(r)(c2)(k)
  val uniqueInGrid1 =
    for(k ← 0 to 8; i ← 0 to 2; j ← 0 to 2;
        r ← 0 to 2; c1 ← 0 to 1; c2 ← c1+1 to 2)
      yield !vars(3*i + r)(3*j + c1)(k) ||
            !vars(3*i + r)(3*j + c2)(k)
  val uniqueInGrid2 =
    for(k ← 0 to 8; i ← 0 to 2; j ← 0 to 2; r1 ← 0 to 2;
        c1 ← 0 to 2; c2 ← 0 to 2; r2 ← r1+1 to 2)
      yield !vars(3*i + r1)(3*j + c1)(k) ||
            !vars(3*i + r2)(3*j + c2)(k)
  val forcedEntries =
    for(r ← 0 to 8; c ← 0 to 8 if sudoku(r)(c) != None)
      yield Or(vars(r)(c)(sudoku(r)(c).get - 1))
  val allConstraints =
    onePerEntry ++ uniqueInColumns ++ uniqueInRows ++
    uniqueInGrid1 ++ uniqueInGrid2 ++ forcedEntries
  solve(And(allConstraints:_*))
}
```

Figure 4.6 – Implementing a Sudoku solver with the CafeSat API.

fresh Boolean variables with the above operations. We implement a structure preserving translation to CNF [PG86]. This transformation avoids the exponential blow up of the naive CNF transformation by introducing a fresh variable for each sub-formula and asserting the equivalence of the new variable with its corresponding sub-formula.

### 4.4.2 Features and Implementation

In this section, we present the architecture and features of CafeSat. We discuss the different heuristics implemented and also describe some of the data structures used. The solving component of CafeSat is currently about 1,300 lines of code. This does not include the API layer. CafeSat is open source and available on GitHub[5].

In general, we avoid recursion and try to use iterative constructs as much as possible. We use native JVM types whenever possible. We rely on mutable data structures to avoid expensive heap allocations. In particular, we make extensive use of arrays with primitive types such as Int and Double. Those types are handled well by the Scala compiler, which is able to map them to the native int[] and double[] on the JVM.

The input (CNF) formula contains a fixed number $N$ of variables, and no further variables are introduced in the course of the algorithm. Thus, we can represent variables by integers from 0 to $N-1$. Many properties of variables such as their current assignments and their containing clauses can then be represented using arrays where the indices represent the variable. This provides a very efficient $O(1)$ mapping relation. Literals are also represented as integers, with even numbers being positive variables and odd numbers being negative variables.

We now detail the important components of the SAT procedure.

**Branching Decision** We rely on the VSIDS decision heuristic, introduced initially by the Chaff SAT solver [MMZ+01]. However, we implement the variation of the heuristic described in MiniSat. We keep variables in a priority queue, sorted by their current VSIDS score. For a branching decision, we extract the maximum element of the queue that is not yet assigned. This is the branching literal.

We use a custom implementation of a priority queue that supports all operations in $O(\log N)$, including a delete by value of the variables (without any use of pointers). The trick is to take advantage of the fact that the values stored in the heap are integers from 0 to $N-1$, and to maintain an inverse index to their current position in the heap. The heap is a simple binary heap built with an array. In fact, we store two arrays, one for variables and one for their corresponding score. Having two separate arrays seems to be more efficient than having one array of tuples.

[5] https://github.com/regb/cafesat

**Boolean Constraint Propagation** CafeSat implements the two-watched literals described by the Chaff paper. We implement a custom `LinkedList` to store the clauses that are currently watching a literal. An important feature of our implementation is the possibility of maintaining a pointer to elements we might need to remove, so that a remove operation can be done in $O(1)$ while iterating over the clauses. This is a typical use-case for the two-watched literal, where we need to traverse all clauses that are currently watching the literal, to find a new literal to watch, and to add the current clause to the watchers of the new literal while removing it from the previous one. All operations need to be very fast because they are done continuously on all unit propagation steps.

**Clause Learning** In the original DPLL algorithm, the exhaustive search was explicit, successively setting each variable to true and false after exploring the subtree. A more recent technique consists in doing conflict analysis and then learning a clause before backtracking. The intuition is that this learnt clause is a reason the search was not able to succeed in this branch. This learning scheme also enables the solver to do long backtracking, returning to the first literal choice that caused the clause to be unsatisfiable, instead of the most recent one.

In CafeSat, we implement a conflict-analysis algorithm for learning new clauses. For this, we use the 1UIP learning scheme [ZMMM01]. We also apply clause minimization as invented by MiniSat. We use a stack to store all assigned variables and to maintain a history. We also store for each variable the clause (if any) responsible for its propagation. This implicitly stores the implication graph used in the conflict analysis.

**Clause Deletion** To select which clauses to keep and which ones to drop, we use an activity-based heuristic similar to the one used for decision branching. We set a maximum size to our set of learnt clauses and, whenever we cross this threshold, we delete the clauses with the worst activity score. To ensure completeness and termination, we periodically increase this threshold.

Our current implementation simply stores a list of clauses and sorts them each time we need to remove the least active ones. We assume that clause deletion only happens after a certain number of conflicts, so it is not a very frequent operation. Besides, it could be cheaper to sort the list only each time it is needed, rather than to maintain the invariant in a priority queue for each operation.

**Restarting Strategy** We use a restart strategy, based on a starting interval that slowly grows over time. The starting interval is $N$, which is the number of conflicts until a restart is triggered. A restart factor $R$ will increase the interval after each restart. This increases in the restart interval guarantees completeness of the solver. In the current implementation, $N = 32$ and $R = 1.1$.

| **Version** | *naive* | | *counters* | | *conflict* | | *two-watched* | | *minimization* | | *optimization* | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | Succ. | Time | Succ. | Time | Succ. | Time | Succ. | Time | Succ. | Time | Succ. | Time |
| **uf20** | 100 | 0.171 | 100 | 0.046 | 100 | 0.085 | 100 | 0.090 | 100 | 0.052 | 100 | 0.052 |
| **uf50** | 100 | 0.171 | 100 | 0.127 | 100 | 0.325 | 100 | 0.336 | 100 | 0.084 | 100 | 0.081 |
| **uuf50** | 100 | 0.507 | 100 | 0.179 | 100 | 0.658 | 100 | 0.701 | 100 | 0.111 | 100 | 0.095 |
| **uf75** | 100 | 3.948 | 100 | 0.444 | 100 | 1.170 | 100 | 1.320 | 100 | 3.138 | 100 | 0.122 |
| **uf100** | 30 | 27.05 | 99 | 4.006 | 91 | 7.567 | 93 | 5.844 | 100 | 0.225 | 100 | 0.183 |
| **uuf100** | 44 | 25.42 | 94 | 10.81 | 45 | 25.06 | 53 | 18.24 | 100 | 0.369 | 100 | 0.275 |
| **uf125** | 0 | NA | 55 | 18.73 | 43 | 20.07 | 52 | 18.02 | 100 | 0.393 | 100 | 0.317 |
| **uf200** | 0 | NA | 0 | NA | 7 | 28.30 | 7 | 28.48 | 60 | 6.688 | 100 | 2.131 |
| **uf250** | 0 | NA | 0 | NA | 0 | NA | 0 | NA | 22 | 25.46 | 64 | 16.01 |

Figure 4.7 – Comparing the performance of CafeSat across several versions and optimizations.

### 4.4.3 Experiments

We ran a set of experiments to evaluate the effect of various optimizations that have been implemented over the development of CafeSat. The purpose was to gain some insight on how the incremental refinement of a basic SAT solver can lead to a relatively efficient complete solver. We selected a few important milestones in the development of CafeSat, and we compared their performance on a set of standard benchmarks.

Our results are summarized in Figure 4.7. The experiments were run on an Intel core I5-2500K with 3.30GHz and 8 GiB of RAM. A timeout was set to 30 seconds. The running time is shown in seconds. The versions are organized from the most ancient to the most recent; their description is as follows:

**naive.** Based on the straightforward implementation techniques that use ASTs to represent formulas, and that use recursive functions along with pattern matching for DPLL and BCP.

**counters.** Uses specialized clauses. Each variable is associated with adjacency lists of clauses containing the variable. It uses counters to quickly determine whether a clause becomes SAT or leads to a conflict.

**conflict.** Introduces conflict-driven search with clause learning. This is a standard architecture for modern SAT solver. However the implementation at this stage suffers from much overhead.

**two-watched.** Implements the BCP based on two-watched literals.

**minimization.** Focuses on a more efficient learning scheme. The conflict analysis is optimized and the clause learnt is minimized. It also introduces clause deletion.

**optimization.** Applies many low-level optimizations. A consistent effort is invested in avoiding object allocation as much as possible, and overhead is reduced due to the use of native `Arrays` with `Ints`. We implemented dedicated heap and stack data structures, as well as a linked list optimized for our two-watched literal implementation.

| Benchmark | CafeSat | | Sat4j | |
|---|---|---|---|---|
| | % Suc. | Time (s) | % Suc. | Time (s) |
| **uf50** | 100 | 0.0014 | 100 | 0.0008 |
| **uf100** | 100 | 0.0040 | 100 | 0.0032 |
| **uuf100** | 100 | 0.0069 | 100 | 0.0063 |
| **uf125** | 100 | 0.0136 | 100 | 0.0119 |
| **uf200** | 100 | 0.5526 | 100 | 0.2510 |
| **uf250** | 63 | 4.5972 | 100 | 2.3389 |
| **bmc** | 92 | 3.9982 | 100 | 1.4567 |

Figure 4.8 – Comparing the performance of CafeSat vs Sat4j, a mature Java-based SAT solver.

The benchmarks are taken from SATLIB [HS00]. We focus on uniform random 3-SAT instances, as SATLIB provides a good number of them for many different sizes. Thus, we are able to find benchmarks that are solvable even with the very first versions; and this results in better comparisons.

From these results, we can see that the *naive* version is able to solve relatively small problems and has little overhead. However, it is unable to solve any problem of consequent size. The introduction of the conflict analysis (version *conflict*) actually had much overhead in the analysis of the conflict and thus did not bring any performance improvement. The key step is the optimization of this conflict analysis (version *minimization*), this diminishes the overhead on the conflict analysis, thus reducing time spent in each iteration and minimizing the learning clause. Smaller clauses imply more triggers for unit propagation and a better pruning of the search space.

It is somewhat surprising that the addition of the two-watched literal scheme has little effect on the efficiency of the solver. The implementation at that time was based on Scala `List` from the standard library. The *optimization* version introduces dedicated data structure to maintain watcher clauses. These results show that without a carefully crafted implementation, even smart algorithmic optimizations do not always improve performance.

To put some perspective on the performance of CafeSat, we also ran some comparison with a reference SAT solver. We chose Sat4j [BP10] as it is a fast SAT solver written for the JVM. CafeSat (as well as Sat4j) is currently unable to compete with SAT solvers written in C or C++. Thus, our short-term goal will be to match the speed of Sat4j.

The experiments are summarized in Figure 4.8 with the percentage of successes and average times. We set a timeout of 20 seconds. The average time is computed by considering only instances that have not been timed out. We used the most recent version of CafeSat and turned off the restarting strategy. We compared with Sat4j version 2.3.3, that, as of this writing, is the most recent version available. We use a warm-up technique for the JVM, consisting in solving the first benchmark from the set three times before starting the timer. The **bmc** benchmarks are formulas generated by a model checker on industrial instances. They are also standard

problem from SATLIB. They contain up to about 300,000 clauses.

Our solver is competitive with Sat4j on the instances of medium sizes, however it is still a bit slow on the largest instances. That CafeSat is slower than Sat4j should not come as a shock. Sat4j has been under development for more than five years and is considered to be the best SAT solver currently available on the JVM.

### 4.4.4 Towards SMT Solving

A modern SMT solver architecture is based on the generalization of DPLL to arbitrary theories, referred as DPLL(T) [GHN+04]. CafeSat's DPLL engine is a first step towards a more general SMT solving infrastructure.

We started some streams of work on specific theory solvers in CafeSat. Our extensions have not been reported yet, but are available as part of the official CafeSat distribution. To support the theory of uninterpreted functions, we completed a first implementation of an efficient congruence closure [NO07] algorithm. We also implemented a solver for the theory of recursive data types [Opp78, BST07].

Additionally, we generalized our DPLL implementation to a DPLL(T) style architecture, where the theory solver becomes a parameter to the DPLL search. We also integrated CafeSat with a parser for SMT-LIB, giving an interface to solve SMT formulas in either of the above theory. Although this work is still very much in progress, we hope that CafeSat will eventually emerge as an accessible SMT solver and will play an important role in the Leon (and Scala) infrastructure.

# 5 Case Studies

In this chapter, we demonstrate the expressiveness of Leon's language by implementing more ambitious programs. Despite it being only a subset of Scala and having some restrictions, we show that it is possible to implement actual applications. Throughout this thesis, we developed a number of features for supporting a programming environment that ensures safety and still provides good expressiveness.

One of our key design principles is to ensure that a program verified by Leon never crashes at runtime. This has led us to introduce some restrictions in order to ensure this safety; these restrictions also help us to verify the specifications. With the case studies presented in this chapter, we demonstrate that these restrictions are not too limiting, and that verifying advanced programs — by translation to function code — is feasible.

We expand on the development of two applications from distinct domains. By showcasing the working software for each application domain, we give a sense of the potential of our system for helping produce verified and useful applications. Although the programs presented here are not production ready, they still go beyond simple verified API and interact with the underlying system using an input/output interface, and graphical component.

The workflow we use to develop these software is to first run the code through the verification system; and then, once verified, compile the program with a back-end. The programs we present in this chapter target Scala.js and a custom C back-end provided by Leon. Assuming the chosen back-end respects the correct semantics of Scala, the properties that Leon verify will hold when running the program.

When interfacing with the environment (the browser in Scala.js, the file system in the C back-end) we can take two alternative approaches. The first one, which we use in the Scala.js application, is to develop a verified core module in the Leon language, and to write a driver code that makes full use of the features of Scala.js but is not verified by Leon. The second approach, which we use in program targeting the C back-end, is to provide a library abstraction of the system API; this abstraction enables us to verify the entire program in Leon, and to

compile and run the program without modifications. Choosing which approach to use is a matter of convenience: The former works best when the back-end imposes a framework to be respected, whereas the latter is adapted when the back-end is less intrusive and the programmer has better control of the execution flow.

## 5.1 Browser Game: 2048

For a first domain, we look at web browser applications and, in particular, a browser game. We implement a fully working browser game; the game is based on the HTML5 canvas technology and is compiled with Scala.js[1]. We wrote the game logic entirely in the Leon language, and proved a good number of properties. A small non-verified kernel of driver code is used to render the model on the screen and to dispatch user events to the game-logic module. In principle, we could architecture the system such that the Scala.js APIs are abstracted by Leon libraries, hence enabling Leon to be run on the whole program. But, we did not do so, as there are few additional properties that would have been verified this way.

We built a clone of the popular game 2048[2]. You can try out our version of the game online[3]. The code is concise, due to Scala's expressive nature — which Leon partially captures. The verified module is reported in Appendix A.3, and the complete project is available on GitHub[4]. In this section, we review some of the properties that Leon proves about the game.

The game state is represented with a LevelMap that contains the 16 different Cells that can contain the numbers to be merged.

```
case class Cell(var n: Option[BigInt])
class LevelMap
```

We use mutable cells, because we believe this is both a more natural representation of the problem and a more interesting test case for verification. We defined helper methods for specifications on the LevelMap:

```
class LevelMap {
  def content: Set[BigInt]
  def totalPoints: BigInt
  def nbEmptyCells: BigInt
}
```

content returns the numbers on the map as a set, totalPoints returns the sum of elements, and nbEmptyCells returns the number of cells without an element. These methods are used to state properties of the game logic. For example, we implement a method to place a random value on the map:

[1] https://www.scala-js.org/
[2] http://2048game.com/
[3] https://epfl-lara.github.io/verified-2048/
[4] https://github.com/epfl-lara/verified-2048/

```
def setRandomCell(map: LevelMap, v: BigInt)
                 (implicit state: Random.State): Unit = {
  require(map.nbEmptyCells > 0 && (v == 2 || v == 4))
  ...
} ensuring(_ ⇒ {
    map.nbEmptyCells == old(map).nbEmptyCells − 1 &&
    map.totalPoints == old(map).totalPoints + v &&
    map.content == old(map).content + v
})
```

Leon proves that the implementation meets the contract, which gives us a fairly high confidence that the function is correct. The specification is not complete, as it does not check that elements in the map remain in the same positions; but by checking the validity of the content and the total points, it covers several possible implementation errors.

Additionally, we have implemented moveX functions that apply the sliding and merge operations of the game. Here is the signature of the moveLeft:

```
def moveLeft(map: LevelMap): Unit = {
  ...
} ensuring(_ ⇒ {
  map.totalPoints == old(map).totalPoints &&
  noHolesLeftToRight(map)
})
```

Leon verifies that after applying a moveLeft, the sum of all numbers on the map does not change, despite any merge that occurs, and that there are no holes between two numbers, from left to right.

**Limitations of our Model** This development effort helps us identify some limitations in the Leon language, particularly with respect to our treatment of aliasing. Recall that we make a global assumption that, in the same scope, no two pointers share the same mutable objects. In our implementation, Cell is a mutable object, and a LevelMap instance owns their unique pointers. It is forbidden, at any point, to have a pointer to a LevelMap and a pointer to one of the Cells. In particular, we cannot write convenient methods that select individual cells in the map, such as

```
class LevelMap {
  def randomFreeCell(implicit state: Random.State): Cell = {
    ...
  } ensuring(c ⇒ c.isEmpty)
}
```

This method returns a reference to a Cell instance, which would violate our assumption. We work around this issue by writing functions that directly perform the action on the cells; in

this particular case, a random Cell is directly modified, as shown above with the setRandomCell. This is arguably a less clean implementation, but it is nevertheless correct and has solid specifications.

## 5.2 LZW Compression

In the context of a master's thesis that developed a new back-end to Leon [Ant17], an implementation of the Lempel–Ziv–Welch (LZW) compression algorithm was made. Although this example was developed independently of this thesis and as a demonstration of the capabilities of Leon to generate C code, we believe it is a very convincing example of an important building block that relies heavily on the imperative features that we have contributed.

The LZW compression algorithm is a technique for encoding (and decoding) data. Recent versions of this algorithm are still being used today; for example, the GIF format relies on it. Thus, this is an important algorithm that performs a key operation that needs to be correct.

Compression happens to have a nice mathematical description. Indeed, the correctness of an encode operation can be described as

```
def theorem(x: Input): Boolean = {
  decode(encode(x)) == x
} holds
```

Of course, the above does not tell the whole story. For example, according to this specification, the identity function behaves as a valid encoder. Ideally, we want to at least add a constraint on the size of the output of the encoding.

However, the point of this section is not to develop a fully verified state-of-the-art compression algorithm. Such a goal, though worthwhile, is beyond the scope of this thesis. We want to demonstrate that the language of Leon is suitable for implementing important applications; and we believe that LZW compression fits this description. Although the implementation contains a significant number of properties to check, the properties were devised as a mean to help with the implementation, and not for reaching complete functional correctness. The developer of the program reported that Leon was useful in finding several counterexamples to these properties; these counterexample have then been used to identify and fix bugs.

The implementation can be found in Appendix A.4. The program is intended to be run with the C back-end of Leon. This means that there is a main function, and that the code uses the file system to encode/decode an actual file. The file system is provided by the standard library of Leon and uses modeling techniques described in Chapter 3. The implementation makes extensive use of arrays and class definitions. The code uses objects with invariants to wrap arrays and to expose a richer API (see the Buffer and Dictionary classes).

Although Leon times out on a certain number of verification conditions, the fact that Leon does

not find a counterexample gives us some confidence in the correctness of the implementation. We are not aware of any bugs in the implementation; the program has been tested with some actual inputs, with both the regular Scala compiler and after a translation to C.

The timeouts are due to the recursive structure of properties over arrays. Many properties must hold for each element of the array. These properties are expressed with recursive functions that must be unrolled (indefinitely) by Leon. One way to address this issue is with the inclusion of quantifiers into the language [VK16], as a way to more directly express some specifications.

# 6 Related Work

We complete this thesis with a review of related work. We discuss previous work on Leon and how the work in this thesis improves on in. We cover the relevant work on software verification, and position Leon into the landscape of verification tools. We look at progress made in theorem proving, in particular recent progress on SMT solvers that drive the development of Leon. We also review research on aliasing, unique pointers, and effects.

**The Leon System** This thesis extends and improves the Leon verification system for Scala. Leon originated as a solver for a purely functional language with recursive functions over unbounded and recursive data types [SDK10, SKK11]. It was then extended with basic imperative constructs [BKKS13] that are presented in Chapter 2. More recently, we solved the soundness issues related to the treatment of integral data types [BK15]. Recent work also added support for higher-order functions, integrating them into the core solving algorithm [VKK15, VK16]. The support for objects with state, presented in Chapter 3, is in preparation for being published. One data type often used in imperative low-level code that Leon still does not integrate is floating-point numbers. Some recent efforts [DK14, Dar14] have explored this avenue, and, hopefully, will be integrated into Leon in the future.

Leon is also able to check the termination of a program [VK16]. The verification module of Leon assumes termination of functions; it might prove invalid properties on non-terminating functions. As a result, the soundness of Leon is dependent on either the assumption that the program terminates, or on the termination checker's ability to prove the termination of all functions. Another line of work in Leon is the inference of symbolic resource bounds [MK14, MKK17].

Beyond software verification, Leon has been extended to synthesis [MW71, MW80] and repair. The synthesis module of Leon [KKKS13] relies on common features and, in particular, the solver for the functional core. Repair further extends synthesis [KKK15] and is a particularly impressive application of the capabilities of the system.

An inter-procedural effect analysis for general Scala code was proposed in the past but never

fully integrated into Leon [KKS13]; our approach, instead, documents effects through mutability of function argument types.

**Software Verification** Leon shares similarities with interactive theorem provers such as Isabelle [Pau94] and ACL2 [KMM00b]. Leon tends to be more automated in typical use, but less expressive in general. These systems are usually safer as they have their own small internal rule systems to perform proofs, whereas Leon relies on external automated theorem provers.

Tools that provides an expressive programming language similar to Leon include Verifun [WS03], Dafny [Lei10, Lei12], VeriFast[1], and Why3 [FP13]. In particular, Dafny has a similar language with imperative features, is fully automated, and relies on an SMT solver. In addition to different input languages, Dafny internally uses translation into nondeterministic guarded command language, whereas Leon internally works with deterministic functions. Dafny supports a `int` type that represents mathematical integers, as well as `nat` that supports natural numbers, but appears to not have support for bit-vectors, at the time of writing.

Dafny targets BoogiePL, an intermediate language tailored for verification [BCD+05]. Boogie can be compared to PureScala, as it offers a similar architecture: a core language with a complete verification infrastructure, and font-ends that translate source languages into it. Although, the Boogie language differs significantly in its design, as it represents state and is lower level. Boogie directly generates verification conditions from this representation. Viper is another intermediate verification language, inspired by Boogie, but with a focus on handling permission logics [MSS16, SS15].

Verifun [WS03] attempts to automatically prove properties on a small functional programming language. It is a hybrid system that involves the user when the proof fails to complete. Their functional language only provides natural numbers as numerical types. Contrary to Leon, they do not support imperative constructs. Why3 [FP13] is also a mostly automated verification tool based on SMT solvers. The language is a dialect of ML and, much like our solution, it provides two different types for integers and bit-vectors. It also has extensive support for imperative programming, including exceptions. It generates verification conditions by using a standard weakest-precondition procedure, hence differs from our transformation-based approach. SBV is a Haskell package[2] that relies on SMT solvers to solve properties about Haskell program. SBV supports multiple SMT solvers. Compared to Leon, SBV is a lighter abstraction over SMT solvers, using Haskell as a front-end to SMT solvers. SBV does not implement an independent algorithm to handle recursive functions. Several other tools exist for the verification of contracts in functional languages with higher-order reasoning [XJC09, Xu12, THH12].

The standard verification conditions generation relies on the technique of weakest preconditions [Dij76]. Leon generates verification conditions, but as the language is purely functional,

[1] https://people.cs.kuleuven.be/~bart.jacobs/verifast/
[2] https://hackage.haskell.org/package/sbv

it is mostly a direct translation, with the caveat of handling functions. The technique of Weakest precondition, however, proposes a way to generate a mathematical formula that represents imperative programs [Dij76, NL98, GC10]. Our transformation to an intermediate functional language instead delays the encoding to logic and can avoid an exponential growth in the size of formulas generated, similarly to the work of Flanagan et al. [FS01]. Another work has independently presented deductive rules for transforming local imperative programs into recursive functions [Myr08, MG09]. Our transformation, presented in Chapter 2, produces similar shapes of programs, but their work focuses on formalization and proving correctness of the translations, whereas we integrate it into a verification system and make it cohabit with other features.

From the early days of programming, certain languages have been designed with verification in mind. Such programming languages usually have built-in features to express specifications that can be verified automatically by the compiler itself. These languages include Spec# [BLS04], GYPSY [Amb77] and Euclid [LGH+78]. Eiffel [Mey91] popularized design by contract, where contracts are preconditions and postconditions of functions as language annotations. We find that Scala's contract functions, defined in the library, work just as well as built-in language contracts, and they encourage experimenting with further specification constructs [Ode10]. The language F* [SHK+16] is both a proof assistant and a general-purpose programming language designed with verification in mind. It supports dependent types, higher-order functions, and an effect system.

Overflow detection, including value-losing conversions and shift operations, has been studied in the past [BSC+07, DLRA12]. According to some results, the intentional use of wraparound is rather common [DLRA12], and defaulting to an error on every overflow is not necessarily the best approach. In Leon, we can choose to enforce (using an option) a disciplined programming style that forbids all overflows, or to allow the use of overflow and to focus on functional correctness of the function to determine if a wraparound is valid. Although Leon is limited to the detection of integer overflow, it has the advantage of being able to statically prove the absence of overflow, whereas many tools rely on dynamic checking.

**Theorem Proving and Decision Procedures** Leon relies extensively on the existence of efficient automated decision procedures to handle logic formulas. Decision procedures are usually specialized and efficient algorithms, which contrasts with more general theorem provers, that are based on resolution. These provers include Vampire [RV99, KV13], E [Sch13], and iProver [Kor08]. They tend to be very flexible, with the ability to define axioms to describe theories in which to prove theorems. The domain of formulas generated by Leon is fairly standard, and this limits the interest of using this family of solvers.

Interactive theorem provers, including Isabelle [Pau94], ACL2 [KMM00b], and Coq [BC04], share similarities with Leon itself. They provide some form of programming language to state theorem and develop proofs, which is something that has been explored in Leon as well. They

generally evolved from a mathematical background with the purpose of formalizing theorems, whereas Leon came from a programming language with a focus on computing. Recent work has explored using Isabelle as an alternative solver for Leon [HK16].

Leon takes advantage of specialized and efficient decision procedures. At the core of these technologies lies the original DPLL scheme [DP60, DLL62]. However, due to the exponential complexity of the SAT problem, there was a lack of practical applications of DPLL-based solvers. Several optimizations helped make SAT solvers into practical tools [SS96, MMZ+01, ZMMM01]. Most modern SMT solvers build on a generalization of DPLL to arbitrary theory [GHN+04, KG07] and profit from this same optimizations. The solver that we developed, CafeSat, follows this design as well.

Of particular interest to Leon are several theories that help modeling fundamental types and operations in programming languages. The theory of equality with uninterpreted functions [NO80, GHN+04, NO03, NO07] is extremely useful for reasoning abstractly about program functions. Z3 [DMB08] is a very complete SMT solver that supports this theory, as well as linear integer arithmetic, arrays [dMB09], bit-vectors [GD07], and a way to combine several theories together [NO79]. The main competitor to Z3, in terms of coverage of theories, is CVC4 [BCD+11]. Leon integrates both Z3 and CVC4 as available solvers to use for proving verification conditions. Princess is an SMT solver with strong support for linear integer arithmetic [RÖ8]. With its implementation in Scala, there is an opportunity for a tighter integration with Leon, and a recent effort has begun adding support for Princess inside Leon.

**Invariants and Interpolation** The generation of Verification conditions usually relies on the presence of loop invariants in order to generate formulas for programs with loops. Invariants are also related to function contracts, as a contract can be used as an abstraction of a function call when generating a formula. In Leon, this relation is made explicit with our translation from loops to recursive functions, and from invariants to contracts. Leon also does not strictly need loop invariants, as it uses unrolling to lazily expand the whole definition of the loop/-function into the generated formulas. With this unrolling, Leon can discover counterexamples without the need for invariants, just by expanding the definition until it is sufficient to discover the counterexample. Verification conditions generation will however ignore the loop implementation and use its invariant instead, potentially losing the counterexample.

One particular case of loop invariant generations is the problem of deriving worst-case bounds of programs [HSR+00, Fer04, GESL06, GJK09]. We explored, independently of this thesis, the derivation of algebraic symbolic bounds for a restricted form of loops [BHHK10]. These bounds express relations among program variables hence are loop invariants. Leon would benefit from a loop invariant generation module, as invariants are essential in proving the validity of programs.

Craig interpolants [Cra57] can help to justify why a counterexample is spurious. Interpolation can be used in predicate abstraction, as a heuristic to discover new predicates [HJMM04]. It

is also useful in refining the set of predicates for predicate abstraction [JM06] and in loop-invariants generation [McM08],

We explored the use of interpolation for synthesis from components on unbounded domains; we presented our results at FMCAD 2013 [KB13]. This work was not included in the main body of this thesis. It is a line of work towards synthesis, although using interpolation can have applications in verification. The technique is sound, and it is complete for constraints for which an interpolation procedure exists, which includes e.g. propositional logic, bit-vectors, linear integer arithmetic, recursive structures, finite sets, and extensions of the theory of arrays. For the approach to work in practice, we need well-behaved interpolation procedures that prefer simpler and computationally shorter interpolants; a requirement that is, in any case, desirable for interpolation in predicate abstraction refinement [JM06].

Craig interpolants have been generalized to sequence interpolants [JM06] for refining paths in control-flow graph of programs. A further generalization is that of tree interpolants [MA13] that can be used for encoding dependencies between tree-sutructued program paths. Tree interpolants provide a nested structure and therefore enable reasoning about program with function calls. In a work that is not included in the present thesis, we address the problem of extracting tree interpolants in arbitrary first-order logic theories [BGKK13]. Our method was implemented in the Vampire theorem prover [RV99, KV13] hence extended Vampire with new features for theory reasoning and interpolation. We reduced the problem of tree interpolation to iterative applications of Craig interpolation on tree nodes. More details are available in the original work published at LPAR 2013 [BGKK13]. Another generalization of interpolants are disjunctive interpolants [RHK13] that are very suitable to solve Horn clauses.

**Effects** An effect, in general, is any observable actions from outside a private scope. The most obvious kind of effects is the mutations of shared state, but effects can also include memory accesses (read and write), memory allocation (which modifies the heap), and exceptions. In Leon, we limited the definition of effects to visible modifications of shared objects; memory allocations are not relevant, as the language provides high-level abstractions and hides memory management details. Furthermore, the language does not support exceptions.

Effect systems introduce the notion of an effect as a formal object of a programming language [GL86, LG88, NN99, MM09]. An effect system is a form of an extended static type system that associates each expression with its effects in addition to its type. This enables the programmer to explicitly state facts about the effects, thus enabling the compiler to reason on effects in a more principled way and to potentially unlock new optimization opportunities. In particular, effect systems have applications in concurrency [FF00, AFF06, ABHI11]. Leon does not introduce a new notation for declaring effects, but it uses some sort of implicit effect system when it comes to higher-order functions. For parameters that are of a function type, we assume that any mutable type in the arguments of the signature declares an implicit effect on this argument. However, for top-level functions, to determine the effects, we perform a

simple alias analysis; we do not rely solely on the mutability of the type of the parameters. Although traditional applications of effects include optimizations, in the present work, we are only concerned by the guarantees they bring for our reduction. Another relevant line of work is the work on algebraic effects handlers [PP03, PP13, Lei16] that provide a rich way to model effects.

A framework for polymorphic effect systems for Scala with minimal syntactic overhead was developed [ROH12, Ryt13]. This system addresses the issue of verbosity with respect to higher-order functions that are polymorphic on the effects of their arguments. Leon, to some extent, also suffers from transitive effects coming from a function parameter; but the notation is already extremely lightweight, as a mutable parameter of a function value is implicitly an effect. Leon does not have a built-in way to declare polymorphic effects, although the way we encode closures at the end of Section 1.1 appears to be related; but it uses parameters to declare effects instead of additional annotations. Following a similar strategy, we define a higher-order function with a generic effect:

```
case class Effect[E: Mutable](e: E)

def hof[E: Mutable](f: (Int, Effect[E]) => Int, e: Effect[E]): Int = f(1, e)
```

The higher-order function is essentially polymorphic in the effect of its parameter. It needs to explicitly declare an effect as well, which is done by the second parameter in its signature. We can then use it with a particular instance of a function with effects:

```
case class State(var x1: Int, var x2: Int, var x3: Int)

def incAndAdd(x: Int, effect: Effect[State]): Int = {
  effect.e.x1 += 1
  effect.e.x2 += 1
  effect.e.x3 += 1
  effect.e.x1 + effect.e.x2 + effect.e.x3
}

def test = {
  val state = State(10, 12, 14)
  val res = hof(incAndAdd, Effect(state))
  assert(state.x1 == 11)
  assert(state.x2 == 13)
  assert(state.x3 == 15)
  assert(res == 11 + 13 + 15)
}
```

Although the end result is slightly verbose, we are still able to abstract over effects. Our work obviously differs as it only targets simple side-effects and not generic effects. An extension of that generic framework [ROH12] presents a modular effect system for checking purity of Scala

programs [RAO13].

Alternatively to using effect systems, trying to derive effects by using automated methods that build on top of pointer analysis methods [Bar78, Ban79, CBC93, SR05] has been done. These approaches have the advantages of automation and minimal annotations. However, they do not bring the additional expressive power that effect systems do with their ability to document and validate the intended effects. This lack of explicit intent means that the programmer might not be aware of the consequences that a small modification of the code can cause, such as a loss of performance due to a missed optimization. In addition, due to the transitive nature of effects, in order to be precise, an effect analysis need to consider the whole program, which comes at the cost of separate compilation. Effect systems are inherently more modular as functions can expose effects in their interface. A possible trade-off is to still perform separate compilation but to be more conservative about the effects of external functions. In Leon, we make a similar conservative assumption about the effects of first-class functions, as we do not always have access to the implementation. However, we assume that we have access to the entire program and are thus able to derive precise effects of top-level functions.

An inter-procedural effect analysis is also available for general Scala code [KKS13]. In Leon, we designed, instead, a small integrated effect analysis to support our usage of mutability to document effects.

**Controlling Aliasing** We designed our input language with some strong restrictions on aliasing and sharing. Lines of work in controlling aliasing typically have the objective to optimize the program by re-using memory cells, such as in destructive updates. Other applications include the potential to parallelize computations on non-shared values. By using static analysis methods for implicitly deriving aliasing information, some of these benefits can be obtained automatically. Our interest in understanding the aliasing of objects in Leon is limited to ensuring the correctness of our transformation rules. However, we still build on existing work, and discuss how our treatment compares to the state of the art.

Our aliasing rules draw inspirations from linear types [Gir87, Wad90, WW01, FD02] that originated in linear logic [Gir87] and were studied originally by Wadler [Wad90]. A value with a linear type must be used exactly once — it cannot be duplicated and cannot be discarded. One selling point of linear types is to optimize memory allocation and deallocation for linear values. As the creation and destruction points are known statically, there is no need for garbage collection. In this thesis, we do not study how to compile Leon code, and this particular advantage is not relevant to this work. In particular, the “no discard” rule does not serve any purpose in our system, and we do not maintain it. Wadler also mentions that, for destructive updates, “pure linearity . . . is a stronger constraint than necessary”, and that we only need to guarantee uniqueness when actually updating a value of linear type. The introduction of the `let!` construct permits the user to perform multiple reads on a linear value, also called an observer [Ode92] in this state, before setting the value to a linear type. Fundamentally, the

semantics of linear type is *single-used*, whereas our semantics is rather *single-assigned*.

The mix of linear types with nonlinear in the type system offers a practical complement to another linear language from Lafont [Laf88], in which every single value has exactly one reference to it. In Leon, we also introduce two distinct families of types, with mutable and non-mutable types, where non-mutable types allow sharing and mutable types limit it.

Hogg studied how to protect against aliasing in object-oriented languages [Hog91]. Objects, and in particular references, complicate the picture, thus marking a local variable as read-only does not necessarily make the actual object read-only. Hogg proposes that the variables and results of methods marked explicitly as `read` were to be protected from being assigned to variables, ensuring that their values cannot be modified through aliasing. In Leon, variables and objects are by default read-only, and mutability is explicitly annotated. Despite not having a `read` annotation for mutable objects, which prevents updating value, we can achieve a similar goal by using specification with `old` keyword.

Minsky [Min96] uses the concept of *unique* pointers for references to unique objects, objects that should not be shared. He proposes to modify the semantics of assigning unique pointers into a *move* operation instead. In Leon, mutable pointers are automatically unique, instead of being optionally annotated. Although the current iteration of Leon prevents the assignments of mutable values, we explore the possibility of passing ownership from pointers to pointers, similar to the move semantics. The optional annotation of *non-consumable* parameters is in some sense the default behavior of parameters in Leon. Uniqueness is guaranteed by nullifying pointers at runtime. Alias burying [Boy01a] proposes to handle uniqueness through existing language features instead and gives a modular static analysis. Substantial additional work has been done on uniqueness types for aliasing [BNR01, Boy01b, BLS03]. The challenges of properly handling the uniqueness of an aggregate of objects [CW03] is related to how we forbid aliasing to parts of a mutable objects. Also related is work on access permissions in Chalice [HLMS13] and the line of work on separation logic [Rey02]. In the context of Scala, a type system where uniqueness is enforced by using capabilities has been proposed [HO10]. A more recent work [HL16] proposed controlling aliasing in Scala with a concept of passing capabilities with implicit parameters to access shared data.

Some programming languages integrate a notion of ownership and aliasing at the level of the type system. Rust [rus16] is a recent system language with an advanced type system — in particular supporting affine types and regions — with the purpose of achieving safety properties. It uses unique pointers and borrowed references as a way to explicitly control aliasing. In contrast, the subset that Leon supports also gives the option of purely functional programming with immutable data and unrestricted sharing.

Automated analysis methods can infer unique types without any annotation from the programmer. This is sometimes more convenient, as the techniques can be used to automatically optimize the program, and not restrict the type system. However, uniqueness constraints on pointers are implicit, hence if a reference is not actually unique, it would simply be ignored by

an optimizing compiler, and not marked as an error. Several people have studied such methods to automatically and statically detect destructive updates [DB76, Hud86, BHY89, WR99].

# Conclusions

In this thesis, we have contributed to the development of an infrastructure for developing safe programs in Scala. We have given an overview of the Leon verification system and have extensively presented the current state of the input language of Leon. We have discussed and explained the design decisions we made in order to identify a subset of Scala that is both expressive and automatically verifiable. We have further demonstrated that this language was useful, by building non-trivial software in this subset. Due to the natural expressive power of Scala — which our subset captures — we have been able to create useful systems, including a browser-based game, and to verify a certain number of their properties; these systems can actually be compiled and ran (played). We believe that we have found a sweet spot for a language that is extremely safe and that supports automated verification, and still provides enough expressiveness to be able to efficiently code in. Our experiments have shown that we are on the right path.

We have devised new algorithms to extend the Leon system with support for additional features required by our design. We have followed a general architecture inspired from compilers, with a pipeline of transformation and simplification phases. This architecture has proven very helpful in sustaining the continuous development of the Leon system, and in ensuring a clean integration of the many features and language constructs. Our main contributions in this thesis have been to develop several of these phases.

We have presented transformation rules that eliminate local state and that reduce imperative algorithms to purely functional algorithms. Our transformation assumes local state only, with functions still being pure. Keeping the state local has proven useful for maintaining a clean and modular interface, while allowing for, sometimes, more optimal and natural implementations. Our results have demonstrated that we can maintain the exact semantics of the original imperative code by using only a functional code, hence we have completely separated the solver implementation from having to reason about the imperative semantics.

We have also shown how to handle mutable types and side-effects. We have rewritten functions with side-effects into purely functional functions, with a new signature that is explicit about the function effects. We have described how to restrict the type system of the language to ensure equivalence under the presence of aliasing. We were able to translate specifications on mutable state, including class invariants, to prove them in the functional core of Leon.

The combination of these two transformations can transform away all stateful elements of the input program and reduce it to an equivalent, purely functional, program. We have been able to implement multiple algorithms that mix both functional styles and imperative paradigm with mutable objects, and we were able to prove many properties in our system.

Finally, we have presented our work on bridging the Leon language with SMT solvers. We have explained how to soundly encode certain primitive data types of Leon into logic formulas. We have built a modern and efficient SAT solver in Scala, which has helped us explore the trade-offs of using a high-level language for building high-performance tools that require correctness. Our experience has shown that it is possible to reach satisfying performance, but at the cost of sacrificing certain higher-level features. Furthermore we have described the abstraction we build on top of the SMT-LIB standard, and how we have used it to integrate Leon with multiple solvers.

## Further Work

**Expanding the Language** One avenue of future work is the further extension of the input language. We already addressed some low-hanging fruit in Section 3.4, but to close the gap with Scala there are several more challenging features that are still missing from the Leon language.

With our restrictions for aliasing, we reject many programs that would have been valid in Scala. Although some of them arguably use excessive aliasing, we acknowledge that aliasing is generally useful. We should explore ways to relax these aliasing restrictions, along the lines of the extensions discussed in Section 3.4.2. We outlined some techniques to maintain mappings between aliases and the original identifiers. These techniques are a way to extend the flexibility of a semantic that maintains unique pointers. Alternatively, we could depart from this uniqueness constraint and permit the sharing of mutable values. In this case, we can draw from research in alias analysis and automatically derive alias information. Using this information, we can apply our rewriting only when the analysis determines it is safe to do so; otherwise we can fall back to a more general, and less efficient, encoding.

In this thesis, we have ignored the interaction of state with object orientation. We have limited our treatment of objects to simple mutable data structures. The possibility of storing state in an abstract class, encapsulated behind overridable methods, does not fit well into our current translation. Object orientations offer several interesting challenges, such as dynamic dispatch that requires a form of pointer analysis, and encapsulation that hides fields that used to be explicit under our translation into a copy operation. Furthermore, abstract types with type instantiations in subclasses offer a rich modeling power, but they also raise new questions about how to integrate these features in the current framework.

We can also broaden the support for primitive types: Besides `Int`, Scala has the types `Byte`, `Short`, and `Long`; they behave just as the type `Int` but with different bit sizes. A parallel work has

begun integrating bytes into Leon [Ant17], and it would be straightforward to generalize to any of the other bit-vector types. A floating-point arithmetic standard is also emerging for SMT solvers [BTRW15], and this would open the possibility of supporting Float and Double types in the language as well, thus completing the support of important primitive types of Scala and integrating previous work on real numbers [DK14, Dar14].

**Infrastructure** Although there is steady progress in state-of-the-art SMT solvers, we do not have direct control over the direction of this research. We are planning to build on the CafeSat solver and eventually use it as one of the standard SMT solvers of Leon. CafeSat is implemented entirely in Scala, follows a modern DPLL(T) architecture, and is easily extensible with new theories.

In this thesis, we have stressed the fact that Leon programs can be run by the standard Scala infrastructure. Although we have not touched on this in the thesis, Leon also embeds its own evaluator and bytecode generator for the JVM. A recent master's thesis [Ant17] (that we have supervised) has also begun the exploration of an new back-end — a subset of C. This subset has applications in embedded systems and seems to be a promising fit for a significant part of the Leon language. In particular, we are able to compile mutable objects and arrays in a way that uses only stack allocations and no dynamic memory allocations. The efforts made so far demonstrate potential for using Leon as a way to write safe and verified low-level code. We consider the integration of further features into this back-end, in order to make better use of the expressiveness of Scala.

The existence of back-end infrastructures within Leon, whether being a bytecode generator for the JVM or a code generator targeting an external compiler, opens up the possibility of using the Leon toolset to perform optimizations on the program. These optimizations can combine the constraint solving infrastructure of Leon to achieve advanced analysis and eventually generate more efficient code. One interesting avenue is on using compact bit-vectors to store data. It is common to use Ints for tasks such as counting iterations in a for-loop; although what we actually meant was the semantic of mathematical integers, BigInts. Some early experiments have shown that a simple loop counter implemented with BigInt instead of Int can lead to orders of magnitude of difference in performance. The promise to code with the correct type — BigInt — while still achieving the performance of the efficient type — Int — is extremely enticing and is definitely worth pursuing in the future.

## Final Words

Today's society relies on software more than ever before. Medical devices, airplanes, banks, and even cars . . . they all depend on a heavy stack of complex programs. The correctness of these programs is of the utmost importance to our safety; the most minor of bugs can have deadly consequences, potentially leading to billions of dollars in losses, or even worse, human deaths. We, the programming community, have the duty to ensure that these systems are

built with the best practices available. For too many years, we have relied mostly on testing to achieve this goal. It has been a bumpy ride, but despite some accidents along the way, the software have held up.

But, when I look into the future — no matter in which direction — the only thing I see is software. We live in the Age of Information. An age where most things move at the speed of light. An age where people simply cannot keep up with the amount of data being processed. An age where computers rule. This is our age, and in it, software will eventually be ubiquitous. So, let me ask you a question. Are you willing to bet our future on *most-of-the-time* correct programs? I believe we should aim at certainty — mathematical certainty — and we should pursue this holy grail — complete and automatic program verification. It is my hope that this thesis has brought a piece to the edifice of software verification, no matter how tiny it is. *Testing has carried us this far; now is time for verification to carry the torch forward.*

# A Verified Source Code

## A.1 Tutorial

```
import leon.lang._
import leon.lang.StaticChecks._

object Lists {

  abstract class List[A]
  case class Cons[A](head: A, tail: List[A]) extends List[A]
  case class Nil[A]() extends List[A]


  //def sizeOverflow[A](l: List[A]): Int = (l match {
  // case Nil() => 0
  // case Cons(_, t) => 1 + sizeOverflow(t)
  //}) ensuring(res => res >= 0)

  def sizeSpec[A](l: List[A]): BigInt = (l match {
    case Nil() => BigInt(0)
    case Cons(_, t) => 1 + sizeSpec(t)
  }) //ensuring(res => res >= 0)

  def size[A](l: List[A]): BigInt = {
    var res: BigInt = 0
    var lst: List[A] = l
    (while(!isEmpty(lst)) {
      lst = tail(lst)
      res += 1
    }) invariant(res + sizeSpec(lst) == sizeSpec(l))
    res
  } ensuring(res => res == sizeSpec(l))
```

```
def isEmpty[A](l: List[A]): Boolean = l match {
    case Nil() => true
    case _ => false
}

def tail[A](l: List[A]): List[A] = {
  require(!isEmpty(l))
  l match {
    case Cons(_, t) => t
  }
}


//def foreach

case class BankAccount(var checking: BigInt, var savings: BigInt) {
  require(checking >= 0 && savings >= 0)

  def balance: BigInt = {
    checking + savings
  } ensuring(_ >= 0)

  def save(amount: BigInt): Unit = {
    require(amount >= 0 && amount <= checking)
    checking = checking - amount
    savings = savings + amount
  } ensuring(_ => balance == old(this).balance)
}

def transfer(from: BankAccount, to: BankAccount, amount: BigInt): Unit = {
  require(amount >= 0 && from.checking >= amount)
  from.checking -= amount
  to.checking += amount
} ensuring(_ => from.balance + to.balance == old(from).balance + old(to).balance &&
                from.checking == old(from).checking - amount &&
                to.checking == old(to).checking + amount)


case class Transaction(
  operation: (BankAccount, BigInt) => Boolean,
  cancel: (BankAccount, BigInt) => Unit,
  account: BankAccount, amount: BigInt, var executed: Boolean
) {

  def execute(): Boolean = {
    executed = operation(account, amount)
    executed
```

```
  }

  def rollback(): Unit = {
    require(executed)
    cancel(account, amount)
    executed = false
  }
}


def execute(transaction1: Transaction, transaction2: Transaction): Boolean = {
  require(!transaction1.executed && !transaction2.executed)
  if(transaction1.execute()) {
    if(transaction2.execute()) true else {
      transaction1.rollback()
      false
    }
  } else false
} ensuring(executed => (
  (executed ==> (transaction1.executed && transaction2.executed)) &&
  (!executed ==> (!transaction1.executed && !transaction2.executed))
))


def addOp(acc: BankAccount, amount: BigInt): Boolean = {
  if(amount < 0) false else {
    acc.checking += amount
    true
  }
}
def subOp(acc: BankAccount, amount: BigInt): Boolean = {
  if(amount < 0 || amount > acc.checking) false else {
    acc.checking -= amount
    true
  }
}

def transfer2(from: BankAccount, to: BankAccount, amount: BigInt): Boolean = {

  execute(
    Transaction((acc, amount) => subOp(acc, amount),
                (acc, amount) => addOp(acc, amount),
                from, amount, false),
    Transaction((acc, amount) => addOp(acc, amount),
                (acc, amount) => subOp(acc, amount),
                to, amount, false)
  )
```

```
  } ensuring(executed =>
      from.balance + to.balance == old(from).balance + old(to).balance &&
      executed ==> (from.checking == old(from).checking - amount &&
                    to.checking == old(to).checking + amount)
    )

  def test(): Unit = {
    val add = Transaction((acc, amount) => {
                            if(amount < 0) false else {
                              acc.checking += amount
                              true
                            }
                          },
                          (acc, amount) => {
                            if(amount < 0 || amount > acc.checking) () else {
                              acc.checking -= amount
                            }
                          },
                          BankAccount(1000, 2000), 500, false)

    add.execute()
    assert(add.account.checking == 1500)
    add.rollback()
    assert(add.account.checking == 1000)
  }


}
```

## A.2 Local State

### A.2.1 Amortized Queue

```
import leon.lang._
import leon.annotation._

object AmortizedQueue {
  sealed abstract class List
  case class Cons(head : Int, tail : List) extends List
  case class Nil() extends List

  sealed abstract class AbsQueue
  case class Queue(front : List, rear : List) extends AbsQueue

  def size(list : List) : BigInt = (list match {
    case Nil() => BigInt(0)
    case Cons(_, xs) => 1 + size(xs)
```

```
}) ensuring(_ >= 0)

def content(l: List) : Set[Int] = l match {
  case Nil() => Set.empty[Int]
  case Cons(x, xs) => Set(x) ++ content(xs)
}

def asList(queue : AbsQueue) : List = queue match {
  case Queue(front, rear) => concat(front, reverse(rear))
}

def concat(l1 : List, l2 : List) : List = {
  var r: List = l2
  var tmp: List = reverse(l1)

  (while(!tmp.isInstanceOf[Nil]) {
    val Cons(head, tail) = tmp
    tmp = tail
    r = Cons(head, r)
  }) invariant(content(r) ++ content(tmp) == content(l1) ++ content(l2) && size(r) +
    size(tmp) == size(l1) + size(l2))

  r
} ensuring(res => content(res) == content(l1) ++ content(l2) && size(res) == size(l1) +
  size(l2))

def isAmortized(queue : AbsQueue) : Boolean = queue match {
  case Queue(front, rear) => size(front) >= size(rear)
}

def isEmpty(queue : AbsQueue) : Boolean = queue match {
  case Queue(Nil(), Nil()) => true
  case _ => false
}

def reverse(l: List): List = {
  var r: List = Nil()
  var l2: List = l

  (while(!l2.isInstanceOf[Nil]) {
    val Cons(head, tail) = l2
    l2 = tail
    r = Cons(head, r)
  }) invariant(content(r) ++ content(l2) == content(l) && size(r) + size(l2) == size(l))

  r
} ensuring(res => content(res) == content(l) && size(res) == size(l))
```

```
  def amortizedQueue(front : List, rear : List) : AbsQueue = {
    if (size(rear) <= size(front))
      Queue(front, rear)
    else
      Queue(concat(front, reverse(rear)), Nil())
  } ensuring(isAmortized(_))

  def enqueue(queue : AbsQueue, elem : Int) : AbsQueue = (queue match {
    case Queue(front, rear) => amortizedQueue(front, Cons(elem, rear))
  }) ensuring(isAmortized(_))

  def tail(queue : AbsQueue) : AbsQueue = {
    require(isAmortized(queue) && !isEmpty(queue))
    queue match {
      case Queue(Cons(f, fs), rear) => amortizedQueue(fs, rear)
    }
  } ensuring (isAmortized(_))

  def front(queue : AbsQueue) : Int = {
    require(isAmortized(queue) && !isEmpty(queue))
    queue match {
      case Queue(Cons(f, _), _) => f
    }
  }

  def enqueueAndFront(queue : AbsQueue, elem : Int) : Boolean = {
    if (isEmpty(queue))
      front(enqueue(queue, elem)) == elem
    else
      true
  } holds

  def enqueueDequeueTwice(queue : AbsQueue, e1 : Int, e2 : Int, e3 : Int) : Boolean = {
    if (isEmpty(queue)) {
      val q1 = enqueue(queue, e1)
      val q2 = enqueue(q1, e2)
      val e1prime = front(q2)
      val q3 = tail(q2)
      val e2prime = front(q3)
      val q4 = tail(q3)
      e1 == e1prime && e2 == e2prime
    } else
      true
  } holds
}
```

### A.2.2 Arithmetic

```
import leon.lang._

object Arithmetic {

  def mult(x : BigInt, y : BigInt): BigInt = ({
    var r: BigInt = 0
    if(y < 0) {
      var n = y
      (while(n != 0) {
        r = r - x
        n = n + 1
      }) invariant(r == x * (y - n) && 0 <= -n)
    } else {
      var n = y
      (while(n != 0) {
        r = r + x
        n = n - 1
      }) invariant(r == x * (y - n) && 0 <= n)
    }
    r
  }) ensuring(_ == x*y)

  def add(x : BigInt, y : BigInt): BigInt = ({
    var r = x
    if(y < 0) {
      var n = y
      (while(n != 0) {
        r = r - 1
        n = n + 1
      }) invariant(r == x + y - n && 0 <= -n)
    } else {
      var n = y
      (while(n != 0) {
        r = r + 1
        n = n - 1
      }) invariant(r == x + y - n && 0 <= n)
    }
    r
  }) ensuring(_ == x+y)

  def addBuggy(x : BigInt, y : BigInt): BigInt = ({
    var r = x
    if(y < 0) {
      var n = y
      (while(n != 0) {
        r = r + 1
```

```
        n = n + 1
      }) invariant(r == x + y - n && 0 <= -n)
    } else {
      var n = y
      (while(n != 0) {
        r = r + 1
        n = n - 1
      }) invariant(r == x + y - n && 0 <= n)
    }
    r
  }) ensuring(_ == x+y)

  def sum(n: BigInt): BigInt = {
    require(n >= 0)
    var r: BigInt = 0
    var i: BigInt = 0
    (while(i < n) {
      i = i + 1
      r = r + i
    }) invariant(r >= i && i >= 0 && r >= 0)
    r
  } ensuring(_ >= n)

  def divide(x: BigInt, y: BigInt): (BigInt, BigInt) = {
    require(x >= 0 && y > 0)
    var r = x
    var q = BigInt(0)
    (while(r >= y) {
      r = r - y
      q = q + 1
    }) invariant(x == y*q + r && r >= 0)
    (q, r)
  } ensuring(res => x == y*res._1 + res._2 && res._2 >= 0 && res._2 < y)

}
```

### A.2.3 Associative List

```
import leon.lang._
import leon.annotation._

object AssociativeList {
  sealed abstract class KeyValuePairAbs
  case class KeyValuePair(key: Int, value: Int) extends KeyValuePairAbs

  sealed abstract class List
  case class Cons(head: KeyValuePairAbs, tail: List) extends List
  case class Nil() extends List
```

```
sealed abstract class OptionInt
case class Some(i: Int) extends OptionInt
case class None() extends OptionInt

def domain(l: List): Set[Int] = l match {
  case Nil() => Set.empty[Int]
  case Cons(KeyValuePair(k,_), xs) => Set(k) ++ domain(xs)
}

def findSpec(l: List, e: Int): OptionInt = l match {
  case Nil() => None()
  case Cons(KeyValuePair(k, v), xs) => if (k == e) Some(v) else find(xs, e)
}

//postcondition should fail as it finds the first
def findLast(l: List, e: Int): OptionInt = {
  var res: OptionInt = None()
  var l2 = l
  while(!l2.isInstanceOf[Nil]) {
    val Cons(KeyValuePair(k, v), tail) = l2
    l2 = tail
    if(k == e)
      res = Some(v)
  }
  res
} ensuring(res => findSpec(l, e) == res)

def find(l: List, e: Int): OptionInt = {
  var res: OptionInt = None()
  var l2 = l
  var seen: List = Nil()
  (while(!l2.isInstanceOf[Nil]) {
    val Cons(head@KeyValuePair(k, v), tail) = l2
    seen = Cons(head, seen)
    l2 = tail
    if(res == None() && k == e)
      res = Some(v)
  }) invariant((res match {
        case Some(_) => domain(seen).contains(e)
        case None() => !domain(seen).contains(e)
      }) && domain(seen) ++ domain(l2) == domain(l))

  res
} ensuring(res => res match {
   case Some(_) => domain(l).contains(e)
   case None() => !domain(l).contains(e)
```

```
  })

  def noDuplicates(l: List): Boolean = l match {
    case Nil() => true
    case Cons(KeyValuePair(k, v), xs) => find(xs, k) == None() && noDuplicates(xs)
  }

  def updateElem(l: List, e: KeyValuePairAbs): List = {
    var res: List = Nil()
    var l2 = l
    var found = false
    val KeyValuePair(ek, ev) = e
    (while(!l2.isInstanceOf[Nil]) {
      val Cons(KeyValuePair(k, v), tail) = l2
      l2 = tail
      if(k == ek) {
        res = Cons(KeyValuePair(ek, ev), res)
        found = true
      } else {
        res = Cons(KeyValuePair(k, v), res)
      }
    }) invariant(
        (if(found)
           domain(res) ++ domain(l2) == domain(l) ++ Set(ek)
         else
           domain(res) ++ domain(l2) == domain(l)
        ))
    if(!found)
      Cons(KeyValuePair(ek, ev), res)
    else
      res
  } ensuring(res => e match {
    case KeyValuePair(k, v) => domain(res) == domain(l) ++ Set[Int](k)
  })

}
```

### A.2.4 List Operations

```
import leon.lang._
import leon.annotation._

object ListOperations {

  sealed abstract class List
  case class Cons(head: Int, tail: List) extends List
  case class Nil() extends List
```

```
sealed abstract class IPList
case class IPCons(head: (Int, Int), tail: IPList) extends IPList
case class IPNil() extends IPList

def content(l: List) : Set[Int] = l match {
  case Nil() => Set.empty[Int]
  case Cons(x, xs) => Set(x) ++ content(xs)
}

def iplContent(l: IPList) : Set[(Int, Int)] = l match {
  case IPNil() => Set.empty[(Int, Int)]
  case IPCons(x, xs) => Set(x) ++ iplContent(xs)
}

def size(l: List) : BigInt = {
  var r: BigInt = 0
  var l2 = l
  (while(!l2.isInstanceOf[Nil]) {
    val Cons(_, tail) = l2
    l2 = tail
    r = r+1
  }) invariant(r >= 0)
  r
} ensuring(res => res >= 0)

def sizeBuggy(l: List) : BigInt = {
  var r: BigInt = -1
  var l2 = l
  (while(!l2.isInstanceOf[Nil]) {
    val Cons(_, tail) = l2
    l2 = tail
    r = r+1
  }) invariant(r >= 0)
  r
} ensuring(res => res >= 0)

def sizeFun(l: List) : BigInt = l match {
  case Nil() => BigInt(0)
  case Cons(_, t) => 1 + sizeFun(t)
}
def iplSizeFun(l: IPList) : BigInt = l match {
  case IPNil() => BigInt(0)
  case IPCons(_, t) => 1 + iplSizeFun(t)
}

def iplSize(l: IPList) : BigInt = {
  var r: BigInt = 0
```

```
    var l2 = l
    (while(!l2.isInstanceOf[IPNil]) {
      val IPCons(_, tail) = l2
      l2 = tail
      r = r+1
    }) invariant(r >= 0)
    r
  } ensuring(_ >= 0)

  def sizeStrongSpec(l: List): BigInt = {
    var r: BigInt = 0
    var l2 = l
    (while(!l2.isInstanceOf[Nil]) {
      val Cons(head, tail) = l2
      l2 = tail
      r = r+1
    }) invariant(r == sizeFun(l) - sizeFun(l2))
    r
  } ensuring(res => res == sizeFun(l))

  def zip(l1: List, l2: List) : IPList = ({
    require(sizeFun(l1) == sizeFun(l2))
    var r: IPList = IPNil()
    var ll1: List = l1
    var ll2 = l2

    (while(!ll1.isInstanceOf[Nil]) {
      val Cons(l1head, l1tail) = ll1
      val Cons(l2head, l2tail) = if(!ll2.isInstanceOf[Nil]) ll2 else Cons(0, Nil())
      r = IPCons((l1head, l2head), r)
      ll1 = l1tail
      ll2 = l2tail
    }) invariant(iplSizeFun(r) + sizeFun(ll1) == sizeFun(l1))

    iplReverse(r)
  }) ensuring(res => iplSizeFun(res) == sizeFun(l1))

  def drunk(l : List) : List = {
    var r: List = Nil()
    var l2 = l
    (while(!l2.isInstanceOf[Nil]) {
      val Cons(head, tail) = l2
      l2 = tail
      r = Cons(head, Cons(head, r))
    }) invariant(sizeFun(r) == 2 * sizeFun(l) - 2 * sizeFun(l2))
    r
  } ensuring (sizeFun(_) == 2 * sizeFun(l))
```

```
  def iplReverse(l: IPList): IPList = {
    var r: IPList = IPNil()
    var l2: IPList = l

    (while(!l2.isInstanceOf[IPNil]) {
      val IPCons(head, tail) = l2
      l2 = tail
      r = IPCons(head, r)
    }) invariant(iplContent(r) ++ iplContent(l2) == iplContent(l) && iplSizeFun(r) +
      iplSizeFun(l2) == iplSizeFun(l))

    r
  } ensuring(res => iplContent(res) == iplContent(l) && iplSizeFun(res) == iplSizeFun(l))

  def reverse(l: List): List = {
    var r: List = Nil()
    var l2: List = l

    (while(!l2.isInstanceOf[Nil]) {
      val Cons(head, tail) = l2
      l2 = tail
      r = Cons(head, r)
    }) invariant(content(r) ++ content(l2) == content(l) && sizeFun(r) + sizeFun(l2) ==
      sizeFun(l))

    r
  } ensuring(res => content(res) == content(l) && sizeFun(res) == sizeFun(l))

  def append(l1 : List, l2 : List) : List = {
    var r: List = l2
    var tmp: List = reverse(l1)

    (while(!tmp.isInstanceOf[Nil]) {
      val Cons(head, tail) = tmp
      tmp = tail
      r = Cons(head, r)
    }) invariant(content(r) ++ content(tmp) == content(l1) ++ content(l2))

    r
  } ensuring(content(_) == content(l1) ++ content(l2))

}
```

## A.3 2048 Clone

```
package leon.game2048

import leon.lang._
import leon.lang.Bag
import leon.annotation._
import leon.lang.StaticChecks._
import leon.util.Random

object Game2048 {

  case class Cell(var n: Option[BigInt]) {
    require(n.forall(v => v >= 0))

    def points: BigInt = n.getOrElse(0)

    def containsPoints: Boolean = n.nonEmpty
    def isEmpty: Boolean = n.isEmpty

    def canMerge(that: Cell): Boolean = that.n.nonEmpty && that.n == this.n

    def emptyAsInt: BigInt = if(n.isEmpty) 1 else 0

    def contentAsBag: Bag[BigInt] = n match {
      case None() => Bag.empty[BigInt]
      case Some(m) => Bag(m -> BigInt(1))
    }
    def content: Set[BigInt] = n match {
      case None() => Set.empty[BigInt]
      case Some(m) => Set(m)
    }
  }

  case class LevelMap(
    c11: Cell, c12: Cell, c13: Cell, c14: Cell,
    c21: Cell, c22: Cell, c23: Cell, c24: Cell,
    c31: Cell, c32: Cell, c33: Cell, c34: Cell,
    c41: Cell, c42: Cell, c43: Cell, c44: Cell
  ) {

    def content: Set[BigInt] = c11.content ++ c12.content ++ c13.content ++ c14.content ++
                               c21.content ++ c22.content ++ c23.content ++ c24.content
                                 ++
                               c31.content ++ c32.content ++ c33.content ++ c34.content
                                 ++
                               c41.content ++ c42.content ++ c43.content ++ c44.content

    def contentAsBag: Bag[BigInt] = c11.contentAsBag ++ c12.contentAsBag ++
```

```
    c13.contentAsBag ++ c14.contentAsBag ++
                                    c21.contentAsBag ++ c22.contentAsBag ++
                                      c23.contentAsBag ++ c24.contentAsBag ++
                                    c31.contentAsBag ++ c32.contentAsBag ++
                                      c33.contentAsBag ++ c34.contentAsBag ++
                                    c41.contentAsBag ++ c42.contentAsBag ++
                                      c43.contentAsBag ++ c44.contentAsBag

  def totalPoints: BigInt =
    c11.points + c12.points + c13.points + c14.points +
    c21.points + c22.points + c23.points + c24.points +
    c31.points + c32.points + c33.points + c34.points +
    c41.points + c42.points + c43.points + c44.points

  def existsEmptyCell: Boolean = c11.isEmpty || c12.isEmpty || c13.isEmpty || c14.isEmpty ||
                                    c21.isEmpty || c22.isEmpty || c23.isEmpty || c24.isEmpty ||
                                    c31.isEmpty || c32.isEmpty || c33.isEmpty || c34.isEmpty ||
                                    c41.isEmpty || c42.isEmpty || c43.isEmpty || c44.isEmpty

  def nbEmptyCells: BigInt = c11.emptyAsInt + c12.emptyAsInt + c13.emptyAsInt +
    c14.emptyAsInt +
                               c21.emptyAsInt + c22.emptyAsInt + c23.emptyAsInt +
                                 c24.emptyAsInt +
                               c31.emptyAsInt + c32.emptyAsInt + c33.emptyAsInt +
                                 c34.emptyAsInt +
                               c41.emptyAsInt + c42.emptyAsInt + c43.emptyAsInt +
                                 c44.emptyAsInt

  def nthFree(n: BigInt): BigInt = {
    require(n < nbEmptyCells)
    var toSkip = n
    var res: BigInt = -1

    if(c11.isEmpty && toSkip == 0 && res == -1) {
      res = 0
    } else if(c11.isEmpty) {
      toSkip -= 1
    }

    if(c12.isEmpty && toSkip == 0 && res == -1) {
      res = 1
    } else if(c12.isEmpty) {
      toSkip -= 1
    }

    if(c13.isEmpty && toSkip == 0 && res == -1) {
      res = 2
```

```
} else if(c13.isEmpty) {
  toSkip -= 1
}

if(c14.isEmpty && toSkip == 0 && res == -1) {
  res = 3
} else if(c14.isEmpty) {
  toSkip -= 1
}

if(c21.isEmpty && toSkip == 0 && res == -1) {
  res = 4
} else if(c21.isEmpty) {
  toSkip -= 1
}

if(c22.isEmpty && toSkip == 0 && res == -1) {
  res = 5
} else if(c22.isEmpty) {
  toSkip -= 1
}

if(c23.isEmpty && toSkip == 0 && res == -1) {
  res = 6
} else if(c23.isEmpty) {
  toSkip -= 1
}

if(c24.isEmpty && toSkip == 0 && res == -1) {
  res = 7
} else if(c24.isEmpty) {
  toSkip -= 1
}

if(c31.isEmpty && toSkip == 0 && res == -1) {
  res = 8
} else if(c31.isEmpty) {
  toSkip -= 1
}

if(c32.isEmpty && toSkip == 0 && res == -1) {
  res = 9
} else if(c32.isEmpty) {
  toSkip -= 1
}

if(c33.isEmpty && toSkip == 0 && res == -1) {
```

```
      res = 10
    } else if(c33.isEmpty) {
      toSkip -= 1
    }

    if(c34.isEmpty && toSkip == 0 && res == -1) {
      res = 11
    } else if(c34.isEmpty) {
      toSkip -= 1
    }

    if(c41.isEmpty && toSkip == 0 && res == -1) {
      res = 12
    } else if(c41.isEmpty) {
      toSkip -= 1
    }

    if(c42.isEmpty && toSkip == 0 && res == -1) {
      res = 13
    } else if(c42.isEmpty) {
      toSkip -= 1
    }

    if(c43.isEmpty && toSkip == 0 && res == -1) {
      res = 14
    } else if(c43.isEmpty) {
      toSkip -= 1
    }

    if(c44.isEmpty && toSkip == 0 && res == -1) {
      res = 15
    } else if(c44.isEmpty) {
      toSkip -= 1
    }

    res
  } ensuring(res => res >= n && res < 16)

  @ignore
  def cells: Vector[Cell] = Vector(c11, c12, c13, c14,
                                   c21, c22, c23, c24,
                                   c31, c32, c33, c34,
                                   c41, c42, c43, c44)
}

/* check that there are no holes in the middle of a row */
def noHoles(c1: Cell, c2: Cell, c3: Cell, c4: Cell): Boolean = {
```

```
    if(c1.isEmpty) c2.isEmpty && c3.isEmpty && c4.isEmpty
    else if(c2.isEmpty) c3.isEmpty && c4.isEmpty
    else if(c3.isEmpty) c4.isEmpty
    else true
  }
  def noMergesOpportunities(c1: Cell, c2: Cell, c3: Cell, c4: Cell): Boolean = {
    !c1.canMerge(c2) && !c2.canMerge(c3) && !c3.canMerge(c4)
  }

  //slide everything to the left, filling empty spaces
  def slideLeft(c1: Cell, c2: Cell, c3: Cell, c4: Cell): Unit = {
    if(c3.isEmpty) {
      c3.n = c4.n
      c4.n = None()
    }
    if(c2.isEmpty) {
      c2.n = c3.n
      c3.n = c4.n
      c4.n = None()
    }
    if(c1.isEmpty) {
      c1.n = c2.n
      c2.n = c3.n
      c3.n = c4.n
      c4.n = None()
    }
  } ensuring(_ =>
    c1.points + c2.points + c3.points + c4.points == old(c1).points + old(c2).points +
      old(c3).points + old(c4).points &&
    noHoles(c1, c2, c3, c4)
  )

  //perform a left slide of the 4 cells. This can be used for any
  //4 celles in any direction, as long as the 4 cells are passed
  //in a coherent order to slideLeft. Merge any required cells
  //together
  def mergeLeft(c1: Cell, c2: Cell, c3: Cell, c4: Cell): Unit = {
    slideLeft(c1, c2, c3, c4)
    if(c3.canMerge(c4)) {
      merge(c4, c3)
    }
    if(c2.canMerge(c3)) {
      merge(c3, c2)
    }
    if(c1.canMerge(c2)) {
      merge(c2, c1)
    }
```

```
  slideLeft(c1, c2, c3, c4)
} ensuring( _ =>
  c1.points + c2.points + c3.points + c4.points == old(c1).points + old(c2).points +
    old(c3).points + old(c4).points &&
  noHoles(c1, c2, c3, c4)
  //noMergesOpportunities(c1, c2, c3, c4)
)

/* check that a left move makes sense (either a hole to fill or a merge opportunity) */
def canSlideLeft(c1: Cell, c2: Cell, c3: Cell, c4: Cell): Boolean = {
  !noHoles(c1, c2, c3, c4) || c1.canMerge(c2) || c2.canMerge(c3) || c3.canMerge(c4)
}
def canMoveLeft(map: LevelMap): Boolean = {
  canSlideLeft(map.c11, map.c12, map.c13, map.c14) ||
  canSlideLeft(map.c21, map.c22, map.c23, map.c24) ||
  canSlideLeft(map.c31, map.c32, map.c33, map.c34) ||
  canSlideLeft(map.c41, map.c42, map.c43, map.c44)
}
def canMoveUp(map: LevelMap): Boolean = {
  canSlideLeft(map.c11, map.c21, map.c31, map.c41) ||
  canSlideLeft(map.c12, map.c22, map.c32, map.c42) ||
  canSlideLeft(map.c13, map.c23, map.c33, map.c43) ||
  canSlideLeft(map.c14, map.c24, map.c34, map.c44)
}
def canMoveRight(map: LevelMap): Boolean = {
  canSlideLeft(map.c14, map.c13, map.c12, map.c11) ||
  canSlideLeft(map.c24, map.c23, map.c22, map.c21) ||
  canSlideLeft(map.c34, map.c33, map.c32, map.c31) ||
  canSlideLeft(map.c44, map.c43, map.c42, map.c41)
}
def canMoveDown(map: LevelMap): Boolean = {
  canSlideLeft(map.c41, map.c31, map.c21, map.c11) ||
  canSlideLeft(map.c42, map.c32, map.c22, map.c12) ||
  canSlideLeft(map.c43, map.c33, map.c23, map.c13) ||
  canSlideLeft(map.c44, map.c34, map.c24, map.c14)
}

//this only merges once, not recursively, thus disprove the final noMergesOpportunities
   postcondition
//def mergeLeftNoRecursive(c1: Cell, c2: Cell, c3: Cell, c4: Cell): Unit = {
// slideLeft(c1, c2, c3, c4)
// if(c3.canMerge(c4)) {
// merge(c4, c3)
// }
// if(c2.canMerge(c3)) {
// merge(c3, c2)
// }
```

```
  // if(c1.canMerge(c2)) {
  // merge(c2, c1)
  // }
  // slideLeft(c1, c2, c3, c4)
  //} ensuring(_ =>
  // c1.points + c2.points + c3.points + c4.points == old(c1).points + old(c2).points +
     old(c3).points + old(c4).points &&
  // noHoles(c1, c2, c3, c4) &&
  // noMergesOpportunities(c1, c2, c3, c4)
  //)

  def moveLeft(map: LevelMap): Unit = {
    require(canMoveLeft(map))
    mergeLeft(map.c11, map.c12, map.c13, map.c14)
    mergeLeft(map.c21, map.c22, map.c23, map.c24)
    mergeLeft(map.c31, map.c32, map.c33, map.c34)
    mergeLeft(map.c41, map.c42, map.c43, map.c44)
  } ensuring(_ =>
    map.totalPoints == old(map).totalPoints &&
    noHoles(map.c11, map.c12, map.c13, map.c14) &&
    noHoles(map.c21, map.c22, map.c23, map.c24) &&
    noHoles(map.c31, map.c32, map.c33, map.c34) &&
    noHoles(map.c41, map.c42, map.c43, map.c44)
  )

  def moveUp(map: LevelMap): Unit = {
    require(canMoveUp(map))
    mergeLeft(map.c11, map.c21, map.c31, map.c41)
    mergeLeft(map.c12, map.c22, map.c32, map.c42)
    mergeLeft(map.c13, map.c23, map.c33, map.c43)
    mergeLeft(map.c14, map.c24, map.c34, map.c44)
  } ensuring(_ =>
    map.totalPoints == old(map).totalPoints &&
    noHoles(map.c11, map.c21, map.c31, map.c41) &&
    noHoles(map.c12, map.c22, map.c32, map.c42) &&
    noHoles(map.c13, map.c23, map.c33, map.c43) &&
    noHoles(map.c14, map.c24, map.c34, map.c44)
  )


  def moveRight(map: LevelMap): Unit = {
    require(canMoveRight(map))
    mergeLeft(map.c14, map.c13, map.c12, map.c11)
    mergeLeft(map.c24, map.c23, map.c22, map.c21)
    mergeLeft(map.c34, map.c33, map.c32, map.c31)
    mergeLeft(map.c44, map.c43, map.c42, map.c41)
  } ensuring(_ =>
```

```
    map.totalPoints == old(map).totalPoints &&
    noHoles(map.c14, map.c13, map.c12, map.c11) &&
    noHoles(map.c24, map.c23, map.c22, map.c21) &&
    noHoles(map.c34, map.c33, map.c32, map.c31) &&
    noHoles(map.c44, map.c43, map.c42, map.c41)
  )

  def moveDown(map: LevelMap): Unit = {
    require(canMoveDown(map))
    mergeLeft(map.c41, map.c31, map.c21, map.c11)
    mergeLeft(map.c42, map.c32, map.c22, map.c12)
    mergeLeft(map.c43, map.c33, map.c23, map.c13)
    mergeLeft(map.c44, map.c34, map.c24, map.c14)
  } ensuring(_ =>
    map.totalPoints == old(map).totalPoints &&
    noHoles(map.c41, map.c31, map.c21, map.c11) &&
    noHoles(map.c42, map.c32, map.c22, map.c12) &&
    noHoles(map.c43, map.c33, map.c23, map.c13) &&
    noHoles(map.c44, map.c34, map.c24, map.c14)
  )


  /*
   * merge `that` into `into`, clearing `that` and setting `into` to
   * the right value.
   */
  def merge(that: Cell, into: Cell): Unit = {
    require(that.n.nonEmpty && that.n == into.n)
    val tmp = that.n.get
    that.n = None()
    into.n = Some(into.n.get + tmp)
  } ensuring(_ => into.points + that.points == old(into).points + old(that).points)



  def setRandomCell(map: LevelMap, v: BigInt)(implicit state: Random.State): Unit = {
    require(map.existsEmptyCell && (v == 2 || v == 4))

    val nbEmptyCells = map.nbEmptyCells
    val randomIndex = leon.util.Random.nextBigInt(nbEmptyCells)
    val realIndex = map.nthFree(randomIndex)

    if(realIndex == 0) {
      assert(map.c11.isEmpty)
      map.c11.n = Some(v)
    } else if(realIndex == 1) {
      map.c12.n = Some(v)
```

```
    } else if(realIndex == 2) {
      map.c13.n = Some(v)
    } else if(realIndex == 3) {
      map.c14.n = Some(v)
    } else if(realIndex == 4) {
      map.c21.n = Some(v)
    } else if(realIndex == 5) {
      map.c22.n = Some(v)
    } else if(realIndex == 6) {
      map.c23.n = Some(v)
    } else if(realIndex == 7) {
      map.c24.n = Some(v)
    } else if(realIndex == 8) {
      assert(map.c31.isEmpty)
      map.c31.n = Some(v)
    } else if(realIndex == 9) {
      assert(map.c32.isEmpty)
      map.c32.n = Some(v)
    } else if(realIndex == 10) {
      assert(map.c33.isEmpty)
      map.c33.n = Some(v)
    } else if(realIndex == 11) {
      assert(map.c34.isEmpty)
      map.c34.n = Some(v)
    } else if(realIndex == 12) {
      assert(map.c41.isEmpty)
      map.c41.n = Some(v)
    } else if(realIndex == 13) {
      assert(map.c42.isEmpty)
      map.c42.n = Some(v)
    } else if(realIndex == 14) {
      map.c43.n = Some(v)
    } else if(realIndex == 15) {
      map.c44.n = Some(v)
    }

  } ensuring(_ => {
    map.nbEmptyCells == old(map).nbEmptyCells - 1 &&
    map.totalPoints == old(map).totalPoints + v &&
    map.content == old(map).content + v
  })


  def popNumber(m: LevelMap)(implicit state: Random.State): Unit = {
    require(m.existsEmptyCell)
    val value: BigInt = 2*(1+leon.util.Random.nextBigInt(2))
    setRandomCell(m, value)
```

```
  }

}
```

## A.4 LZW Compression

```
import leon.lang._
import leon.proof._
import leon.annotation._
import leon.io.{
  FileInputStream => FIS,
  FileOutputStream => FOS,
  StdOut
}

object LZW {

  val DictionarySize = 8192
  val BufferSize = 64

  val AlphabetSize = Byte.MaxValue + -Byte.MinValue

  case class Buffer(val array: Array[Byte], var length: Int) {
    val capacity = array.length
    require(isValid)

    def isValid: Boolean = length >= 0 && length <= capacity && capacity == BufferSize

    def isFull: Boolean = length == capacity

    def nonFull: Boolean = length < capacity

    def isEmpty: Boolean = length == 0

    def nonEmpty: Boolean = length > 0

    def isEqual(b: Buffer): Boolean = {
      if (b.length != length) false
      else { isEmpty || isRangeEqual(array, b.array, 0, length - 1) }
    }

    def size = {
      length
    } ensuring { res => 0 <= res && res <= capacity }

    def apply(index: Int): Byte = {
      require(index >= 0 && index < length)
```

```
    array(index)
  }

  def append(x: Byte): Unit = {
    require(nonFull)
    array(length) = x
    length += 1
  } ensuring { _ => isValid }

  def dropLast(): Unit = {
    require(nonEmpty)
    length -= 1
  } ensuring { _ => isValid && old(this).length == this.length + 1 }

  def clear(): Unit = {
    length = 0
  } ensuring { _ => isEmpty && isValid }

  def set(that: Buffer): Unit = {
    if (that.isEmpty) this.clear
    else setImpl(that)
  } ensuring { _ =>
    that.isValid && this.isValid && isEqual(that) && that.isEqual(old(that))
  }

  private def setImpl(that: Buffer): Unit = {
    require(isValid && that.nonEmpty && that.isValid && capacity == that.capacity)

    val bufferLength = that.length

    length = that.length

    assert(this.isValid)
    assert(this.nonEmpty)
    assert(this.length == that.length)

    var i = 0
    (while (i < bufferLength) {
      array(i) = that.array(i)
      i += 1
    }) invariant {
      that.length == bufferLength && this.length == bufferLength && this.length ==
        that.length &&
      0 <= i && i <= this.length &&
      this.isValid && this.nonEmpty && that.isValid && that.nonEmpty &&
      (i > 0 ==> isRangeEqual(array, that.array, 0, i - 1))
    }
```

```
    assert(that.isValid)
    assert(this.isValid)
    assert(this.nonEmpty)
    assert(isRangeEqual(array, that.array, 0, length - 1))
    assert(this.length == that.length)
    assert(isEqual(that))
  } ensuring { _ =>
    that.isValid && this.isValid && this.nonEmpty &&
    isEqual(that) && that.isEqual(old(that))
  }

}

@inline
def createBuffer(): Buffer = {
  Buffer(Array.fill(BufferSize)(0), 0)
} ensuring { b => b.isEmpty && b.nonFull && b.isValid }

//16-bit code word
case class CodeWord(b1: Byte, b2: Byte)

def index2CodeWord(index: Int): CodeWord = {
  require(0 <= index && index < 65536)
  val signed = index - 32768
  val b2 = signed.toByte
  val b1 = (signed >> 8).toByte
  CodeWord(b1, b2)
}

def codeWord2Index(cw: CodeWord): Int = {
  val signed = (cw.b1 << 8) | (0xff & cw.b2)
  signed + 32768
} ensuring { res => 0 <= res && res < 65536 }

case class Dictionary(val buffers: Array[Buffer], var nextIndex: Int) {
  val capacity = buffers.length
  require(isValid)

  def isValid = 0 <= nextIndex && nextIndex <= capacity &&
                capacity == DictionarySize && allValidBuffers(buffers)

  def isEmpty = nextIndex == 0
  def nonEmpty = !isEmpty
  def isFull = nextIndex == capacity
  def nonFull = nextIndex < capacity
```

```
  def lastIndex = {
    require(nonEmpty)
    nextIndex - 1
  } ensuring { res => 0 <= res && res < capacity }

  def contains(index: Int): Boolean = {
    require(0 <= index)
    index < nextIndex
  }

  def appendTo(index: Int, buffer: Buffer): Boolean = {
    require(0 <= index && contains(index))

    val size = buffers(index).size

    assert(buffer.capacity == BufferSize)
    if (buffer.size < buffer.capacity - size) {
      assert(buffer.nonFull)

      var i = 0
      (while (i < size) {
        buffer.append(buffers(index)(i))
        i += 1
      }) invariant {
        0 <= i && i <= size &&
        (i < size ==> buffer.nonFull)
      }

      true
    } else false
  }

  def insert(b: Buffer): Unit = {
    require(nonFull && b.nonEmpty)

    assert(lemmaSize)
    assert(isValid)
    assert(nonFull)
    assert(nextIndex < capacity)
    assert(nextIndex < DictionarySize)
    assert(nextIndex + 1 <= DictionarySize)

    buffers(nextIndex).set(b)

    assert(lemmaSize)
    assert(isValid)
    assert(nonFull)
```

```
      assert(nextIndex < capacity)
      assert(nextIndex < DictionarySize)
      assert(nextIndex + 1 <= DictionarySize)

      nextIndex += 1
    } ensuring { _ => isValid }

    def encode(b: Buffer): Option[CodeWord] = {
      require(b.nonEmpty)

      var found = false
      var i = 0

      (while (!found && i < nextIndex) {
        if (buffers(i).isEqual(b)) {
          found = true
        } else {
          i += 1
        }
      }) invariant {
        0 <= i && i <= nextIndex && i <= capacity &&
        isValid &&
        (found ==> (i < nextIndex && buffers(i).isEqual(b)))
      }

      if (found) Some(index2CodeWord(i)) else None()
    }
  }

  @inline
  def createDictionary() = {
    Dictionary(Array.fill(DictionarySize){ createBuffer() }, 0)
  } ensuring { res => res.isEmpty }

  def initialise(dict: Dictionary): Unit = {
    require(dict.isEmpty)

    val buffer = createBuffer()
    assert(buffer.isEmpty)

    var value: Int = Byte.MinValue // Use an Int to avoid overflow issues

    (while (value <= Byte.MaxValue) {
      buffer.append(value.toByte) // no truncation here
      dict.insert(buffer)
      buffer.dropLast()
      value += 1
```

```
    }) invariant {
      dict.nonFull &&
      buffer.isEmpty &&
      value >= Byte.MinValue && value <= Byte.MaxValue + 1
    }
  } ensuring { _ => dict.isValid && dict.nonEmpty }

  def encode(fis: FIS, fos: FOS)(implicit state: leon.io.State): Boolean = {
    require(fis.isOpen && fos.isOpen)
    val dictionary = createDictionary()
    initialise(dictionary)
    encodeImpl(dictionary, fis, fos)
  }

  def encodeImpl(dictionary: Dictionary, fis: FIS, fos: FOS)
                (implicit state: leon.io.State): Boolean = {
    require(fis.isOpen && fos.isOpen && dictionary.nonEmpty)

    var bufferFull = false
    var ioError = false

    val buffer = createBuffer()
    assert(buffer.isEmpty && buffer.nonFull)

    var currentOpt = tryReadNext(fis)

    // Read from the input file all its content, stop when an error occurs
    // (either output error or full buffer)
    (while (!bufferFull && !ioError && currentOpt.isDefined) {
      val c = currentOpt.get

      assert(buffer.nonFull)
      buffer.append(c)
      assert(buffer.nonEmpty)

      val code = dictionary.encode(buffer)

      val processBuffer = buffer.isFull || code.isEmpty

      if (processBuffer) {
        // Add s (with c) into the dictionary, if the dictionary size limitation allows it
        if (dictionary.nonFull) {
          dictionary.insert(buffer)
        }

        // Encode s (without c) and print it
        buffer.dropLast()
```

```
        assert(buffer.nonFull)
        assert(buffer.nonEmpty)
        val code2 = dictionary.encode(buffer)

        assert(code2.isDefined)
        ioError = !writeCodeWord(fos, code2.get)

        // Prepare for next codeword: set s to c
        buffer.clear()
        buffer.append(c)
        assert(buffer.nonEmpty)
      }

      bufferFull = buffer.isFull

      currentOpt = tryReadNext(fis)
    }) invariant {
      bufferFull == buffer.isFull &&
      ((!bufferFull && !ioError) ==> buffer.nonEmpty)
    }

    // Process the remaining buffer
    if (!bufferFull && !ioError) {
      val code = dictionary.encode(buffer)
      assert(code.isDefined) // See (*) above.
      ioError = !writeCodeWord(fos, code.get)
    }

    !bufferFull && !ioError
  }

  def decode(fis: FIS, fos: FOS)(implicit state: leon.io.State): Boolean = {
    require(fis.isOpen && fos.isOpen)

    // Initialise the dictionary with the basic alphabet
    val dictionary = createDictionary()
    initialise(dictionary)

    decodeImpl(dictionary, fis, fos)
  }

  def decodeImpl(dictionary: Dictionary, fis: FIS, fos: FOS)
                (implicit state: leon.io.State): Boolean = {
    require(fis.isOpen && fos.isOpen && dictionary.nonEmpty)

    var illegalInput = false
    var ioError = false
```

```
  var bufferFull = false

  var currentOpt = tryReadCodeWord(fis)

  val buffer = createBuffer()

  if (currentOpt.isDefined) {
    val cw = currentOpt.get
    val index = codeWord2Index(cw)

    if (dictionary contains index) {
      bufferFull = !dictionary.appendTo(index, buffer)
      ioError = !writeBytes(fos, buffer)
    } else {
      illegalInput = true
    }

    currentOpt = tryReadCodeWord(fis)
  }

  (while(!illegalInput && !ioError && !bufferFull && currentOpt.isDefined) {
    val cw = currentOpt.get
    val index = codeWord2Index(cw)
    val entry = createBuffer()

    if (dictionary contains index) {
      illegalInput = !dictionary.appendTo(index, entry)
    } else if (index == dictionary.lastIndex + 1) {
      entry.set(buffer)
      entry.append(buffer(0))
    } else {
      illegalInput = true
    }

    ioError = !writeBytes(fos, entry)
    bufferFull = buffer.isFull

    if (!bufferFull) {
      val tmp = createBuffer()
      tmp.set(buffer)
      tmp.append(entry(0))
      if (dictionary.nonFull) {
        dictionary.insert(tmp)
      }

      buffer.set(entry)
    }
```

```
    currentOpt = tryReadCodeWord(fis)
  }) invariant {
    dictionary.nonEmpty
  }

  !illegalInput && !ioError && !bufferFull
}
sealed abstract class Status
case class Success() extends Status
case class OpenError() extends Status
case class EncodeError() extends Status
case class DecodeError() extends Status

implicit def status2boolean(s: Status): Boolean = s match {
  case Success() => true
  case _ => false
}

def _main() = {
  implicit val state = leon.io.newState

  def statusCode(s: Status): Int = s match {
    case Success() => StdOut.println("success"); 0
    case OpenError() => StdOut.println("couldn't open file"); 1
    case EncodeError() => StdOut.println("encoding failed"); 2
    case DecodeError() => StdOut.println("decoding failed"); 3
  }

  def encodeFile(): Status = {
    val input = FIS.open("input.txt")
    val encoded = FOS.open("encoded.txt")

    val res =
      if (input.isOpen && encoded.isOpen) {
        if (encode(input, encoded)) Success()
        else EncodeError()
      } else OpenError()

    encoded.close
    input.close

    res
  }

  def decodeFile(): Status = {
    val encoded = FIS.open("encoded.txt")
```

```
    val decoded = FOS.open("decoded.txt")

    val res =
      if (encoded.isOpen && decoded.isOpen) {
        if (decode(encoded, decoded)) Success()
        else DecodeError()
      } else OpenError()

    decoded.close
    encoded.close

    res
  }

  val r1 = encodeFile()
  statusCode(if (r1) decodeFile() else r1)
}

@extern
def main(args: Array[String]): Unit = _main()

/****** Helpers ******/

def tryReadNext(fis: FIS)(implicit state: leon.io.State): Option[Byte] = {
  require(fis.isOpen)
  fis.tryReadByte()
}

def writeCodeWord(fos: FOS, cw: CodeWord): Boolean = {
  require(fos.isOpen)
  fos.write(cw.b1) && fos.write(cw.b2)
}

def tryReadCodeWord(fis: FIS)(implicit state: leon.io.State): Option[CodeWord] = {
  require(fis.isOpen)
  val b1Opt = fis.tryReadByte()
  val b2Opt = fis.tryReadByte()

  (b1Opt, b2Opt) match {
    case (Some(b1), Some(b2)) => Some(CodeWord(b1, b2))
    case _ => None()
  }
}

def writeBytes(fos: FOS, buffer: Buffer): Boolean = {
  require(fos.isOpen && buffer.nonEmpty)
  var success = true
```

```
    var i = 0

    val size = buffer.size

    (while (success && i < size) {
      success = fos.write(buffer(i))
      i += 1
    }) invariant {
      0 <= i && i <= size
    }

    success
  }

  def isRangeEqual(a: Array[Byte], b: Array[Byte], from: Int, to: Int): Boolean = {
    require(0 <= from && from <= to && to < a.length && to < b.length)
    a(from) == b(from) && {
      if (from == to) true
      else isRangeEqual(a, b, from + 1, to)
    }
  }

  def allValidBuffers(buffers: Array[Buffer]): Boolean = {
    def rec(from: Int): Boolean = {
      require(0 <= from && from <= buffers.length)
      if (from < buffers.length) buffers(from).isValid && rec(from + 1)
      else true
    }
    rec(0)
  }

  /****** Lemmas ******/

  def lemmaSize: Boolean = {
    DictionarySize >= AlphabetSize &&
    BufferSize > 0 &&
    AlphabetSize > 0 &&
    DictionarySize <= 65536 // Cannot encode more index using only 16-bit codewords
  }.holds

  def lemmaBufferFull(b: Buffer): Boolean = {
    b.isFull == !b.nonFull
  }.holds

  def lemmaBufferEmpty(b: Buffer): Boolean = {
    b.isEmpty == !b.nonEmpty
  }.holds
```

```
  def lemmaBufferFullEmpty1(b: Buffer): Boolean = {
    b.isFull ==> b.nonEmpty
  }.holds

  def lemmaBufferFullEmpty2(b: Buffer): Boolean = {
    (BufferSize > 0 && b.isEmpty) ==> b.nonFull
  }.holds

  def lemmaBufferSelfEqual(b: Buffer): Boolean = {
    b.isEqual(b)
  }.holds

  def lemmaBufferEqual(a: Buffer, b: Buffer): Boolean = {
    b.isEqual(a) ==> a.isEqual(b)
  }.holds

  def lemmaBufferEqualImmutable(a: Buffer, b: Buffer): Unit = {
    a.isEqual(b)
  } ensuring { _ => old(a).isEqual(a) && old(b).isEqual(b) }

  def lemmaDictionaryFull(d: Dictionary): Boolean = {
    d.isFull == !d.nonFull
  }.holds

  def lemmaDictionaryFullEmpty(d: Dictionary): Boolean = {
    d.isEmpty ==> d.nonFull
  }.holds

  def lemmaSelfRangeEqual(a: Array[Byte], from: Int, to: Int): Boolean = {
    require(0 <= from && from <= to && to < a.length)
    isRangeEqual(a, a, from, to) because {
      if (from == to) check { lemmaUnitRangeEqual(a, a, from) }
      else {
        check { a(from) == a(from) } &&
        check { a(to) == a(to) } &&
        check { lemmaSelfRangeEqual(a, from, to - 1) } &&
        check { lemmaSelfRangeEqual(a, from + 1, to) }
      }
    }
  }.holds

  def lemmaRangeEqual(a: Array[Byte], b: Array[Byte], from: Int, to: Int): Boolean = {
    require(0 <= from && from <= to && to < a.length && to < b.length)

    ( isRangeEqual(a, b, from, to) ==> isRangeEqual(b, a, from, to) )
  }.holds
```

```
def lemmaSubRangeEqual1(a: Array[Byte], b: Array[Byte], from: Int, to: Int): Boolean = {
  require(0 <= from && from <= to && to < a.length && to < b.length)
  isRangeEqual(a, b, from, to) ==> (
    check { a(from) == b(from) } &&
    check { (from + 1 <= to) ==> isRangeEqual(a, b, from + 1, to) }
  )
}.holds

def lemmaSubRangeEqual2(a: Array[Byte], b: Array[Byte], from: Int, to: Int): Boolean = {
  require(0 <= from && from <= to && to < a.length && to < b.length)
  isRangeEqual(a, b, from, to) ==> (
    check { (a(to) == b(to)) because lemmaRangeEndEqual(a, b, from, to) } &&
    check { (from <= to - 1) ==> isRangeEqual(a, b, from, to - 1) }
  )
}.holds

def lemmaMiniSubRangeEqual1(a: Array[Byte], b: Array[Byte], from: Int, to: Int): Boolean = {
  require(0 <= from && from <= to && to < a.length && to < b.length)
  isRangeEqual(a, b, from, to) ==> (
    isRangeEqual(a, b, from, from) because a(from) == b(from)
  )
}.holds

def lemmaMiniSubRangeEqual2(a: Array[Byte], b: Array[Byte], from: Int, to: Int): Boolean = {
  require(0 <= from && from <= to && to < a.length && to < b.length)
  isRangeEqual(a, b, from, to) ==> (
    isRangeEqual(a, b, to, to) because {
      if (from == to) check { a(to) == b(to) }
      else {
        check { from < to } &&
        check { lemmaMiniSubRangeEqual2(a, b, from + 1, to) } &&
        check { lemmaMiniSubRangeEqual2(a, b, from, to - 1) }
      }
    }
  )
}.holds

def lemmaUnitRangeEqual(a: Array[Byte], b: Array[Byte], pos: Int): Boolean = {
  require(0 <= pos && pos < a.length && pos < b.length)
  isRangeEqual(a, b, pos, pos) ==> (a(pos) == b(pos))
}.holds

def lemmaRangeEndEqual(a: Array[Byte], b: Array[Byte], from: Int, to: Int): Boolean = {
  require(0 <= from && from <= to && to < a.length &&
          to < b.length && isRangeEqual(a, b, from, to))
  ( a(to) == b(to) ) because {
```

```
      if (from == to) trivial
      else {
        check { from < to } &&
        check { lemmaRangeEndEqual(a, b, from + 1, to) }
      }
    }
  }.holds

  def lemmaCodeWordIndexEquality(index: Int): Boolean = {
    require(0 <= index && index < 65536)
    index == codeWord2Index(index2CodeWord(index))
  }.holds


  def lemmaAllValidBuffers(buffers: Array[Buffer]): Boolean = {
    allValidBuffers(buffers)
  }.holds

}
```

# Bibliography


[ABHI11] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33(1):2:1–2:50, January 2011.

[AFF06] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, March 2006.

[Amb77] Allen L. Ambler. GYPSY: A language for specification and implementation of verifiable programs. In *Language Design for Reliable Software*, pages 1–10, 1977.

[Ant17] Marco Antognini. Extending Safe C Support in Leon. Master's thesis, EPFL, 2017.

[App98] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM.

[Ban79] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 29–41, New York, NY, USA, 1979. ACM.

[Bar78] Jeffrey M. Barth. A practical interprocedural data flow analysis algorithm. *Commun. ACM*, 21(9):724–736, September 1978.

[BB09] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[BCD+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[BGKK13] Régis Blanc, Ashutosh Gupta, Laura Kovács, and Bernhard Kragl. Tree interpolation in vampire. In *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, pages 173–181, 2013.

[BHHK10] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. Abc: algebraic bound computation for loops. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 103–118, Berlin, Heidelberg, 2010. Springer-Verlag.

[BHY89] Adrienne Bloss, P. Hudak, and J. Young. An optimising compiler for a modern functional language. *Comput. J.*, 32(2):152–161, April 1989.

[BK15] Régis Blanc and Viktor Kuncak. Sound reasoning about integral data types with a reusable SMT solver interface. In *Scala Symposium*, 2015.

[BKKS13] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An overview of the Leon verification system: Verification by translation to recursive functions. In *Scala Workshop*, 2013.

[Bla12] Régis William Blanc. Verification of Imperative Programs in Scala. Master's thesis, EPFL, 2012.

[Bla13] Régis Blanc. Cafesat: A modern sat solver for scala. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 2:1–2:4, New York, NY, USA, 2013. ACM.

[BLS03] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, 2003.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. pages 49–69. Springer, 2004.

[BNR01] John Boyland, James Noble, and William Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In *ECOOP*, number 2072 in LNCS, 2001.

[Bou92] Raymond T. Boute. The euclidean definition of the functions div and mod. *ACM Trans. Program. Lang. Syst.*, 14(2):127–144, April 1992.

[Boy01a] John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice & Experience*, 6(31):533–553, May 2001.

[Boy01b] John Boyland. The interdependence of effects and uniqueness. In *3rd workshop on Formal Techniques for Java Programs*, 2001.

[BP10] Daniel Le Berre and Anne Parrain. The Sat4j Library, Release 2.2. *JSAT*, 7(2-3), 2010.

[BSC+07] David Brumley, Dawn Xiaodong Song, Tzi-cker Chiueh, Rob Johnson, and Huijia Lin. RICH: automatically protecting against integer-based vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*, 2007.

[BST07] Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. *Electron. Notes Theor. Comput. Sci.*, 174(8):23–37, June 2007.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[BTRW15] Martin Brain, Cesare Tinelli, Philipp Ruemmer, and Thomas Wahl. An automatable formal semantics for ieee-754 floating-point arithmetic. In *Proceedings of the 2015 IEEE 22Nd Symposium on Computer Arithmetic*, ARITH '15, pages 160–167, Washington, DC, USA, 2015. IEEE Computer Society.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 232–245, New York, NY, USA, 1993. ACM.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, December 2011.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

[Cra57] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[CW03] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, pages 176–200, 2003.

[Dar14] Eva Darulova. *Programming with Numerical Uncertainties*. PhD thesis, EPFL, 2014.

[DB76] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Inf.*, 6(1):41–60, March 1976.

[DdM06] Bruno Dutertre and Leonardo M. de Moura. The Yices SMT solver, 2006.

[Dij76] Edsger Wybe Dijkstra. *A discipline of programming*. Prentice-Hall, Englewood Cliffs, N.J, 1976.

[DK14] Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2014.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Commun. ACM*, 5(7), July 1962.

[DLRA12] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 760–770, Piscataway, NJ, USA, 2012. IEEE Press.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[dMB09] Leonardo de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures, 2009.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3), July 1960.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.

[FD02] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, 2002.

[Fer04] C. Ferdinand. Worst case execution time prediction by static program analysis. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 125–, April 2004.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM.

[Fil99] Jean-Christophe Filliâtre. Proof of imperative programs in type theory. In *Selected Papers from the International Workshop on Types for Proofs and Programs*, TYPES '98, pages 78–92, London, UK, UK, 1999. Springer-Verlag.

[Fil03] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program.*, 13(4):709–745, July 2003.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP*, 2013.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205, 2001.

[GC10] Mike Gordon and Hélène Collavizza. Forward with Hoare. In A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 102–121. Springer, 2010.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[GESL06] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 57–66, Dec 2006.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast Decision Procedures. In Rajeev Alur and DoronA. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[GJK09] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 375–385, New York, NY, USA, 2009. ACM.

[GL86] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 28–38, New York, NY, USA, 1986. ACM.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 232–244, New York, NY, USA, 2004. ACM.

[HK16] Lars Hupel and Viktor Kuncak. Translating scala programs to isabelle/hol. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*, pages 568–577, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[HL16] Philipp Haller and Alex Loiko. Lacasa: Lightweight affinity and object capabilities in scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 272–291, New York, NY, USA, 2016. ACM.

[HLMS13] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. Abstract read permissions: Fractional permissions without the fractions. In *VMCAI*, 2013.

[HO10] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 354–378, Berlin, Heidelberg, 2010. Springer-Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hog91] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 271–285, New York, NY, USA, 1991. ACM.

[HS00] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. IOS Press, 2000.

[HSR+00] Christopher Healy, Mikael Sjödin, Viresh Rustagi, David Whalley, and Robert Van Engelen. Supporting timing analysis by automatic bounding of loopiterations. *Real-Time Syst.*, 18(2/3):129–156, May 2000.

[Hud86] Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 351–363, New York, NY, USA, 1986. ACM.

[JLS09] Susmit Jha, Rhishikesh Limaye, and Sanjit Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Computer Aided Verification*, pages 668–674. 2009.

[JM06] Ranjit Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *Proceedings of the 12th International Conference on Tools and*

*Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 459–473, Berlin, Heidelberg, 2006. Springer-Verlag.

[Kas06] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Proceedings of the 14th International Conference on Formal Methods*, FM'06, pages 268–283, Berlin, Heidelberg, 2006. Springer-Verlag.

[KB13] Viktor Kuncak and Régis Blanc. Interpolation for synthesis on unbounded domains. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 93–96, 2013.

[KG07] Sava Krstić and Amit Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *Proceedings of the 6th international symposium on Frontiers of Combining Systems*, FroCoS '07, Berlin, Heidelberg, 2007. Springer-Verlag.

[KKK15] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive program repair. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 217–233, 2015.

[KKKS13] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 407–426, New York, NY, USA, 2013. ACM.

[KKS11] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of Z3: Integrating SMT and programming. In *CADE*, pages 400–406, 2011.

[KKS13] Etienne Kneuss, Viktor Kuncak, and Philippe Suter. Effect analysis for programs with callbacks. In *Fifth Working Conference on Verified Software: Theories, Tools and Experiments*, 2013.

[KMM00a] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[KMM00b] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[Kne16] Etienne Kneuss. *Deductive Synthesis and Repair*. PhD thesis, EPFL, Lausanne, 2016.

[Kor08] Konstantin Korovin. iprover - an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pages 292–298, 2008.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, pages 1–35, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[Laf88] Y. Lafont. The linear abstract machine. *Theor. Comput. Sci.*, 59(1-2):157–180, July 1988.

[LBR09] Daniel Le Berre and Pascal Rapicault. Dependency Management for the Eclipse Ecosystem: Eclipse p2, Metadata and Resolution. In *Proceedings of the 1st international workshop on Open component ecosystems*, IWOCE '09, New York, NY, USA, 2009. ACM.

[Lea07] Gary T. Leavens. Tutorial on jml, the java modeling language. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 573–573, New York, NY, USA, 2007. ACM.

[Lei10] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[Lei12] K. Rustan M. Leino. Developing verified programs with Dafny. In *HILT*, pages 9–10, 2012.

[Lei16] Daan Leijen. Algebraic effects for functional programming. Technical report, August 2016.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[LGH+78] Ralph L. London, John V. Guttag, James J. Horning, Butler W. Lampson, James G. Mitchell, and Gerald J. Popek. Proof rules for the programming language Euclid. *Acta Inf.*, 10:1–26, 1978.

[LM04] K. Rustan M. Leino and Peter Müller. *Object Invariants in Dynamic Contexts*, pages 491–515. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[MA13] Kenneth L. McMillan and Rybalchenko Andrey. Solving constrained horn clauses using interpolation. Technical report, MSR, 2013.

[Mcc62] J. Mccarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[McM08] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *Proceedings of the Theory and Practice of Software, 14th International*

*Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 413–427, Berlin, Heidelberg, 2008. Springer-Verlag.

[Mey86] Bertrand Meyer. Design by contract. Technical report, Interactive Software Engineering Inc., 1986.

[Mey91] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, 1991.

[MG09] Magnus O. Myreen and Michael J. C. Gordon. Transforming programs into recursive functions. *Electron. Notes Theor. Comput. Sci.*, 240:185–200, July 2009.

[Min96] Naftaly H. Minsky. Towards alias-free pointers. In *Proceedings of the 10th European Conference on Object-Oriented Programming*, ECCOP '96, pages 189–209, London, UK, UK, 1996. Springer-Verlag.

[MK14] Ravichandhran Madhavan and Viktor Kuncak. Symbolic resource bound inference for functional programs. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 762–778, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[MKK17] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based resource verification for higher-order functions with memoization. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 330–343, New York, NY, USA, 2017. ACM.

[MM09] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 39–50, New York, NY, USA, 2009. ACM.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, New York, NY, USA, 2001. ACM.

[MSS16] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[MW71] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, March 1971.

[MW80] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, January 1980.

[Myr08] Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2008.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *PLDI*, pages 333–344, 1998.

[NN99] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the Occasion of His Retirement from His Professorship at the University of Kiel)*, pages 114–136, London, UK, UK, 1999. Springer-Verlag.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[NO03] Robert Nieuwenhuis and Albert Oliveras. Congruence closure with integer offsets. In *Logic for Programming, Artificial Intelligence, and Reasoning, 10th International Conference, LPAR 2003, Almaty, Kazakhstan, September 22-26, 2003, Proceedings*, pages 78–90, 2003.

[NO07] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, April 2007.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Ode92] Martin Odersky. Observers for linear types. In *Symposium Proceedings on 4th European Symposium on Programming*, ESOP'92, pages 390–407, London, UK, UK, 1992. Springer-Verlag.

[Ode10] Martin Odersky. Contracts for scala. In *RV*, pages 51–57, 2010.

[Opp78] Derek C. Oppen. Reasoning about recursively defined data structures. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 151–157, New York, NY, USA, 1978. ACM.

[Pau94] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[PG86] David A. Plaisted and Steven Greenbaum. A Structure-preserving Clause Form Translation. *J. Symb. Comput.*, 2(3), September 1986.

[PP03] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[PP13] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[RÖ8] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '08, pages 274–289, Berlin, Heidelberg, 2008. Springer-Verlag.

[RAO13] Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, FTfJP '13, pages 4:1–4:7, New York, NY, USA, 2013. ACM.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74, 2002.

[RHK13] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for horn-clause verification (extended technical report). *CoRR*, abs/1301.4973, 2013.

[ROH12] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 258–282, Berlin, Heidelberg, 2012. Springer-Verlag.

[rus16] The Rust programming language, 2016. https://doc.rust-lang.org/book/ (retrieved 2016-08-01).

[RV99] Alexandre Riazanov and Andrei Voronkov. Vampire. In *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, pages 292–296, 1999.

[Ryt13] Lukas Rytz. *A Practical Effect System for Scala.* PhD thesis, EPFL, 2013.

[Sch13] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[SDK10] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, 2010.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 256–270, New York, NY, USA, 2016. ACM.

[SJP12] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, May 2012.

[SKK11] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS*, pages 298–315, 2011.

[SR05] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 199–215, Berlin, Heidelberg, 2005. Springer-Verlag.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP: a New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, Washington, DC, USA, 1996. IEEE Computer Society.

[SS15] M. Schwerhoff and A. J. Summers. Lightweight Support for Magic Wands in an Automatic Verifier. In J. T. Boyland, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 37 of *LIPIcs*, pages 614–638. Schloss Dagstuhl, 2015.

[Sut12] Philippe Suter. *Programming with Specifications*. PhD thesis, EPFL, Lausanne, 2012.

[THH12] Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In *OOPSLA*, pages 537–554, 2012.

[VK16] Nicolas Voirol and Viktor Kuncak. Automating Verification of Functional Programs with Quantified Invariants. Technical report, EPFL, 2016.

[VKK15] Nicolas Voirol, Etienne Kneuss, and Viktor Kuncak. Counter-example complete verification for higher-order functions. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala 2015, Portland, OR, USA, June 15-17, 2015*, pages 18–29, 2015.

[Wad90] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North Holland, 1990.

[War02] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 187–206, New York, NY, USA, 1999. ACM.

[WS03] Christoph Walther and Stephan Schweitzer. About VeriFun. In *CADE*, pages 322–327, 2003.

[WW01] David Walker and Kevin Watkins. On regions and linear types. In *ICFP*, September 2001.

[XJC09] Dana N. Xu, Simon L. Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In *POPL*, pages 41–52, 2009.

[Xu12] Dana N. Xu. Hybrid contract checking via symbolic simplification. In *PEPM*, pages 107–116, 2012.

[ZKR08] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, Piscataway, NJ, USA, 2001. IEEE Press.

# RÉGIS BLANC

*Curriculum Vitæ*

## EDUCATION

| | |
|---|---|
| September, 2012 - Present | **École Polytechnique Fédérale de Lausanne (EPFL)**, Switzerland<br>*PhD, Computer Science, Advisor: Viktor Kuncak* |
| September, 2010 - July, 2012 | **École Polytechnique Fédérale de Lausanne (EPFL)**, Switzerland<br>*M.Sc., Computer Science, GPA:* 5.64/6.0 |
| August, 2009 - April, 2010 | **University of California, Berkeley**, United States<br>*Visiting Student* |
| September, 2006 - July, 2009 | **École Polytechnique Fédérale de Lausanne (EPFL)**, Switzerland<br>*B.Sc., Computer Science, GPA:* 5.49/6.0 |

## PROFESSIONAL EXPERIENCE

| | |
|---|---|
| February, 2014 - May, 2014 | **IBM Research, Zurich, Switzerland** - *Intern*<br>Developed a software model checker for LLVM IR. Worked directly with LLVM API to integrate into the compiler pipeline. |
| July, 2012 - August, 2012 | **TU Wien, Vienna, Austria** - *Intern*<br>Worked on the Vampire theorem prover. Devised and implemented an algorithm to compute tree interpolants. Published a conference article describing the technique and implementation. |
| August, 2010 - February, 2011 | **NEC Corporation, CCIL, Nara, Japan** - *Intern*<br>Research work on co-creative programming. Developed a web based system, using javascript, for creating, editing, sharing, and playing online tower defense games. |

## PUBLICATIONS

- R. Blanc, V. Kuncak. *Sound reasoning about integral data types with a reusable SMT solver interface.* SCALA'15 Proceedings of the 6th ACM SIGPLAN Symposium on Scala.
- R. Blanc, A. Gupta, L. Kovács, B. Kragl *Tree Interpolation in Vampire* 19th International Conference, LPAR-19, 2013
- V. Kuncak, R. Blanc. *Interpolation for synthesis on unbounded domains.* FMCAD 2013.
- R. Blanc. *CafeSat: A Modern SAT Solver for Scala.* SCALA'13 Proceedings of the 4th Workshop on Scala.
- R. Blanc, V. Kuncak, E. Kneuss and P. Suter. *An Overview of the Leon Verification System.* SCALA'13 Proceedings of the 4th Workshop on Scala.
- R. Blanc, T. Henzinger, T. Hottelier and L. Kovacs. *ABC: Algebraic Loop Bound Computation.* Lecture Notes in Artificial Intelligence (LNAI) 6355, 2011, pp. 103-118.

## PROJECTS

**CafeSat: A Modern SAT Solver for Scala**
Implementation of *CafeSat*, a modern SAT solver in Scala that includes many state-of-the-art methods such as 2-watched literals, conflict-driven clause learning, and random restart.

**Android Mobile Games**
Developed several mobile games for the Android platform. Published the games on Google Play under my personal developer account *RegB*. Games are implemented in Scala, with my own custom game engine. I produced some of the art using the tools GIMP and Inkscape.

**TimeStash: A Social Network for Nostalgia**
Worked, with a small team, on the *TimeStash* social network. Had the role of a full-stack developer, and also participated to the business and marketing process. Built the entire back-end with Scala and the Liftweb framework.

**Verification of Imperative Programs in Scala**, *Master thesis*
Worked on the Leon static verification system for Scala. Implemented an extension to reduce imperative program verification to functional program verification. Implementation language was Scala. Under Prof. Viktor Kuncak and Dr. Philippe Suter at LARA, EPFL.

**Auto-vectorization Pass in LLVM**, *Semester project*
Implemented a state of the art auto-vectorization algorithm. The code is integrated as a pass in the open source compiler framework LLVM. Under Prof. Paolo Ienne and Dr. David Novo Bruna at the Processor Architecture Laboratory (LAP) of EPFL.

**Algebraic bound computation for loops**, *Semester project*
Designed and implemented, in Scala, a new algorithm to analyze the algorithmic complexity of a restricted class of loops. Under Prof. Thomas A. Henzinger and Dr. Laura Kovacs at the Models and Theory of Computation laboratory (MTC) of EPFL. The work was accepted for publication at LPAR.

## TECHNICAL SKILLS

- **Programming Languages:** C, C++, Haskell, Java, OCaml, Scala, Scheme, Python
- **Web Development:** (X)HTML, CSS, PHP, JavaScript
- **Databases:** PostgreSQL, Mysql, Oracle
- **Tools:** Eclipse, Vim, LaTeX

## RELEVANT COURSEWORK

| | |
|---|---|
| Adv. Algorithms | Adv. Compiler Construction |
| Adv. Database | Adv. Topics in Computer Systems |
| Adv. Topics in Programming Language | Artificial Inteligence |
| Computer Aided Verification | Computer Architecture |
| Computer Graphics | Computer Networks |
| Distributed Computer Science | Foundation of Software |
| Graph Theory | Mathematical Logic |
| Network Security | Synthesis, Analaysis, and Verification |

## PERSONAL INFORMATION

- **Languages :** *French* (Native language), *English* (Fluent, C2)
- **Hobbies :** Soccer (playing for 15 years), Tennis, Close up magic